Diplomarbeit zum Thema

Entwicklung eines Editors zum Entwurf von Benutzerschnittstellen für Web Services auf Basis der abstrakten UI-Beschreibungssprache WSGUI

zur Erlangung des akademischen Grades
Diplom-Informatiker

an der Technischen Universität Dresden

Josef Spillner
14. September 2006

Hochschullehrer: Prof. Dr. rer. nat. habil. Dr. h.c. Alexander Schill
Betreuer: Dr.-Ing. Iris Braun

Aufgabenstellung: Mit der Entwicklung der Web-Service-Technologie wurde eine universelle Schnittstelle zur Kopplung von verschiedensten Anwendungen über das Internet geschaffen. Durch die Einbindung von Web Services verschiedener Anbieter in ein Portal ist eine flexible Abbildung von komplexen Geschäftsprozessen möglich. Um eine hohe Flexibilität der Lösung zu ermöglichen, ist eine dynamische Integration der Dienste notwendig. Für die Darstellung im Portal müssen neben der Schnittstellenbeschreibung WSDL zusätzliche Informationen über die Benutzerschnittstelle des Web Service bekannt sein. Diese können mit der am Lehrstuhl entwickelten abstrakten Beschreibungssprache WSGUI festgelegt werden. Um den Entwickler bei der Erstellung dieser WSGUI-Beschreibung für einen vorhandenen Web Service zu unterstützen, wäre ein grafischer Editor wünschenswert, welcher im Rahmen dieser Arbeit entwickelt und prototypisch implementiert werden soll.

Hiermit erkläre ich die vorliegende Diplomarbeit selbständig und ausschließlich unter Verwendung der im Literaturverzeichnis aufgeführten Veröffentlichungen und sonstigen Informationsquellen verfasst zu haben.

Josef Spillner, Dresden, 14. September 2006

_____

Kurzfassung/Abstract

Diese Diplomarbeit behandelt das Themengebiet der automatischen Erzeugung von grafischen Benutzeroberflächen (GUIs) im Kontext von Webservices. Es geht dabei konkret um einen Editor, mit dem Hinweise zur Generierung von Dialogen erstellt werden können. Diese Hinweise sollen dann von den dialogerzeugenden Anwendungen einbezogen werden, um eine dynamische Interaktion mit Webservices durch beliebige Nutzer ohne dienstspezifische Software zu ermöglichen. Die Arbeit führt in die aktuellen Techniken zur GUI-Generierung ein und stellt Transformationsprinzipien vor, die eine Generierung ausgehend von einem formalen Datenmodell ermöglichen. Dabei müssen mangelnde Modellparameter in Beschreibungen von Webservices manuell ausgeglichen werden (WSGUI-Konzepte). Diese Zielstellung führt zum Entwurf des WSGUI-Editors. Die anschließende Implementierung berücksichtigt darüber hinaus Integrationsaspekte wie die Publizierung von WSGUI-Hinweisen, Einbindung von GUI-Übersetzungen und eine Vorschau auf die zu generierenden Dialoge. Abgeschlossen wird die Arbeit mit einer Bewertung des Editors, aber auch einer Reihe von Vorschlägen zur Vereinfachung ähnlich gelagerter zukünftiger Projekte im Umfeld von Webservices.

This thesis covers the topic of auto-generated graphical user interfaces (GUIs) in the context of web services. Its main contribution is a graphical editor which can be used to create hints for dialog generation. Such hints are to be used by dialog-creating applications in order to facilitate a dynamic interaction between a service and a user without the need for special client software. The work introduces current GUI generation techniques and presents principles on transforming formal models into GUIs. Missing model properties in web service descriptions have to be supplied manually – leading to the concept of the WSGUI editor. Its implementation then focuses on the publishing of WSGUI hints, translation thereof and a dialog preview functionality. The thesis is concluded with a review of the editor, but also a couple of suggestions aimed at simplified web service-related development.

Einordnung der Arbeit gemäß *ACM Computing Classification System* (1998): H4.3 [Communications Applications], H5 [Information Interfaces and Presentation], H1.2 [User/-Machine Systems], I7.2. [Document Preparation]

Inhaltsverzeichnis

1. Einführung

Die vorliegende Arbeit ist vor dem Hintergrund entstanden, dass IT-Infrastrukturen mehr und mehr durch lose Kopplung der beteiligten Prozesse gekennzeichnet sind. Jene Prozesse lassen sich als Softwarekomponenten verstehen, oder auch als Dienste, welche über ein Netzwerk miteinander kommunizieren. Dabei spielt es weniger eine Rolle, auf welcher Basis die jeweiligen Dienste implementiert sind, solange sie sich auf der Grundlage eines gemeinsamen Protokolls verständigen können. Durch den Einsatz normierter Beschreibungssprachen für die Schnittstellen der Dienste und den zwischen ihnen transportierten Nachrichten hat sich eine Klasse von Diensten herausgebildet, die als Webservices (WS) bezeichnet werden. Auf deren Basis lassen sich serviceorientierte Architekturen (SOA) konstruieren, indem Webservices bei Bedarf aufgerufen und miteinander verknüpft werden [50][4].

Das Aufrufen der Dienste geschieht meist über spezielle Programme, welche anwendungsspezifische Daten zu einem Dienst senden und das Ergebnis des Aufrufs wieder entsprechend verarbeiten können. Dabei gibt es keinen Grund, warum man nicht auch über ein generisches Programm beliebige Dienste aufrufen könnte. Eine in einem späteren Kapitel vorgestellte Analyse der Nachrichtenstrukturen lässt die Idee zu, dass mit relativ wenig Aufwand ausgehend von den Beschreibungen der Dienstschnittstellen Formulare für die Eingabe der Daten und die Anzeige der Ergebnisse generiert werden können.

Zwar handelt es sich um GUI-Generierung aus einem formalen Modell, für das Verständnis der Konzepte hingegen ist es besser, zuerst mit einem Überblick über den Begriff GUI zu beginnen und dann zu erkunden, wie bereits erprobte Gestaltungskonzepte aus einem Schema heraus abgebildet werden können. Es wäre fatal, etwaige Unzulänglichkeiten in den Schemabeschreibungssprachen in minderqualitative GUIs münden zu lassen.

Die aktuell gebräuchlichen Technologien zur Erzeugung von grafischen Benutzeroberflächen bildet dann auch den Einstieg in diese Arbeit in Kapitel zwei. Dabei werden insbesondere Dialogkonzepte hervorgehoben, da diese für die Interaktion mit Webservices benötigt werden. Eine Auswahl von deklarativen Beschreibungen von Benutzeroberflächen wird hiernach getroffen und deren Anpassbarkeit auf die Dialogkonzepte nachgeprüft.

Das dritte Kapitel beginnt am anderen Ende des Spektrums mit der Vorstellung von Schemasprachen und den üblichen Beschreibungsformaten für Webservices, die ihrerseits Schemasprachen für die Spezifikation des Aufbaus von Ein- und Ausgabenachrichten verwenden. Auf deren Basis werden Interaktionsmuster zwischen Nutzern und Webservices aufgezeigt. Bereits in diesem Kapitel wird dann folgerichtig der Brückenschlag zu einer XML-basierten Formularbeschreibungssprache namens XForms vollführt, und gezeigt, inwiefern sich das Dialogprinzip im Webservicebereich umsetzen lässt. Die Vorstellung von WSGUI-Konzepten und verwandter Arbeiten zur Interaktion mit Webservices erfolgt in diesem Zusammenhang. Insbesondere die Vorgänge der GUI-Generierung aus Schema-

ta im Rahmen der modellgetriebenen Entwicklung [29] werden dazu beschrieben. Eine experimentelle Zusammenstellung und statistische Auswertung vorhandener Webservice-Beschreibungen dient der Validierung derartiger Generierungsideen.

Nach diesen theoretischen Vorarbeiten widmet sich das Folgekapitel der Konzeption des Editors. Neben der Wahl der konkreten Implementierungstechniken spielt hierbei vor allem eine Rolle, inwiefern die vorgestellten WSGUI-Konzepte überhaupt in den Editor eingebunden werden müssen, und wie eine solche Einbindung auszusehen hat. Verwandte Arbeiten auf dem Gebiet des Editor-Entwurfs fließen dabei in die Konzeption mit ein.

Da im Rahmen der Arbeit ein brauchbarer WSGUI-Editor entstanden ist, wird dieser anschließend sowohl von der Architektur her als auch unter Gesichtspunkten der Nutzung vorgestellt. Dabei wird nicht nur auf die Implementierung eingegangen, sondern auch auf die Anbindung an existierende Software zur Bildung eines durchgängigen Autorenprozesses.

Dem Einsatz im Bereich Webservices, aber auch in anderen Anwendungsdomänen, ist dabei ein weiteres Kapitel gewidmet, um die Einsatztauglichkeit und Flexibilität von WSGUI-Konzepten im Sinne einer universell verwendbaren Methodik der GUI-Generierung aus Schemata zu demonstrieren.

Das siebente und letzte Kapitel fasst den Verlauf der Entwicklung zusammen und bewertet die entstandene Lösung. Abgeschlossen wird die Arbeit mit einem Ausblick auf zukünftige Entwicklungen, wobei insbesondere Hinweise auf zukünftige Revisionen der existierenden XML-Standards, welche im Webservice-Umfeld dominieren, gegeben werden. Wenige Änderungen an bestimmten Stellen würden dabei bereits signifikant die GUI-Generierung vereinfachen können und somit die Herausbildung zukünftiger WSGUI-Ansätze beschleunigen.

2. Erzeugung grafischer Bedienoberflächen

Um einen Computer bedienen zu können, benötigen Menschen eine Schnittstelle, die einerseits eine Eingabe von Zielstellungen und andererseits eine Ausgabe von Resultaten der zum Ziel führenden Operationen erlaubt. Die Umsetzung der Schnittstellen involviert Hardware wie Monitore, Maus und Tastatur, aber auch Software, die sogenannten Bedienoberflächen oder Benutzerschnittstellen (*user interfaces*). Die durch Oberflächen angesteuerte Software kann entweder auf dem lokalen Computer oder auf entfernten Rechnern laufen. Für die Belange dieser Arbeit wird von einem entfernt aufgerufenen Dienst ausgegangen. Die Unterscheidung ist für dieses Kapitel jedoch noch nicht relevant, so dass die Erzeugung von Bedienoberflächen relativ eigenständig präsentiert werden kann. Dennoch wird an einigen Stellen die Eignung als Schnittstelle für Dienste hervorgehoben.

Die Entwicklung einer ansprechenden und funktionellen visuellen Schnittstelle für eine Anwendung (inklusive der Ansteuerung von Diensten) wird traditionell „von Hand" durchgeführt, und ist dabei auf die jeweilige Anwendung fixiert. Ändern sich deren Parameter, so muss unter Umständen die gesamte Schnittstelle erneuert werden. Wünschenswert ist hingegen eine möglichst automatische Erzeugung der Bedienoberfläche, die auch über die Grenzen einer Anwendung oder eines Dienstes hinaus dynamisch in anderen Szenarien einsetzbar ist [5].

Zur Generierung einer solchen Oberfläche muss die Analyse der momentanen Möglichkeiten in die zu verwendende Zielplattform einerseits und die zugrunde liegende Sprache zur Spezifikation andererseits unterteilt werden. Um keine starre Schnittstelle zu erhalten, sondern auch Interaktivität und Rückmeldungen zum Benutzer zu ermöglichen, schließt sich hieran eine Betrachtung der funktionellen Elemente an.

2.1. Plattformen und Bibliotheken

Der Dialog eines Benutzers mit einer Anwendung hat seit Jahrzehnten traditionell über speziell entwickelte Bedienoberflächen stattgefunden. Diese bieten grundlegende Elemente zur Ein- und Ausgabe von Text an, oft unterstützt von grafischen Symbolen und Hilfetexten, sowie Navigationselemente zur dialoggestützten Kommunikation. Sekundäre Elemente wie Aktivitätsanzeigen, Hervorhebungen von fehlerhaften Eingaben oder auch automatische Vervollständigungen sind nicht unbedingt notwendig, erhöhen jedoch deutlich den Komfort und sind aus Gründen der Akzeptanz beim Benutzer für die Evaluation mit einzubeziehen. Schließlich dürfen auch moderne softwareergonomische Gesichtspunkte wie Barrierefreiheit, Internationalisierung und Integration in die Arbeitsumgebung nicht vernachlässigt werden.

2.1.1. Arbeitsplatzumgebung (Desktop)

Die weitverbreitetste Schnittstelle zwischen Benutzern und ihren Anwendungen sind die aus den Terminal-Systemen der 60er Jahre hervorgegangenen grafischen Bedienoberflächen, auch als Desktops bezeichnet. Viele verbreitete Arbeitsschritte haben sich seit jener Zeit nicht geändert, jedoch kamen und gingen mehrere Generationen von Technologien, um Desktopoberflächen ansprechend und effizient zu gestalten. Die wichtigsten Vertreter werden dabei kurz angesprochen werden, um einen Überblick zu erhalten.

Evolution von Desktopumgebungen

Zur Anfangszeit der bildschirmbasierten Arbeit kamen Textoberflächen, so genannte *Text User Interfaces (TUI)* zum Einsatz. Diese konnten neben regulären alphanumerischen Zeichen und Sonderzeichen auch speziell gestaltete Zeichen wie senkrechte Linien oder Ecken darstellen, deren Kombination es erlaubte, einen Bildschirm in Unterfenster zu teilen sowie Menüs und Laufbalken hinzuzufügen. Vertreter dieser Gattung sind `ncurses`-basierte Oberflächen auf Unix-Systemen sowie das Äquivalent der DOS-Betriebssystemfamilie. Nach wie vor finden sich viele derartige Systeme im öffentlichen Einsatz als Kassensysteme und in Arztpraxen, da die Dialogführung für die Erfüllung der dort gegebenen Aufgaben ausreicht. Man spricht dabei auch von Masken, die oftmals an Datenbanken angebunden sind, und teilweise sogar automatisch auf Basis des jeweiligen Datenbankschemas automatisch generiert werden [33]. Auch das in Deutschland verbreitete Videotext-System lässt sich hier einordnen.

Mit der Verbreitung von Personalcomputern (PCs) Ende der 80er Jahre sind pixelbasierte Oberflächen, die *Graphical User Interfaces* (*GUIs*) im eigentlichen Sinne, entstanden. Angaben über die Auflösung und die Farbtiefe gehörten zum Standardrépertoire der Anwendungsentwicklung. Beschränkte man sich zu Beginn noch auf VGA-Ausgabe (640x480 Pixel bei 16 Farben), so ist nunmehr eine Auflösung von 1024x768 Pixeln bei „voller Farbtiefe" (etwa 16,8 Millionen Farben) als gängig anzusehen. Ein bekanntes Problem bei Desktopanwendungen sind dabei Dialoge, deren Ausmaße bei bestimmten Auflösungen über den Desktop hinausragen. Eine GUI-Generierung sollte auch hierfür geeignete Abhilfen schaffen können, wobei das Problem durch sogenannte GUI-Builder, also Entwurfswerkzeugen für die interaktive Gestaltung von Oberflächen, eigentlich schon behoben sein sollte [59]. Die „Hall of Shame" [38] der GUI-Gestaltung zeigt aber deutlich, dass durch die Freiheiten eines GUI-Builders sämtliche Anforderungen an die Ergonomie übergangen werden können.

Mit der Zunahme leistungsstarker Computer und hochauflösender Bildschirme gibt es nunmehr eine dritte Generation, die sogenannten *Zoomable User Interfaces (ZUI)*. Hierbei erfolgt die Berechnung der Koordinaten nicht pixelbasiert, sondern in einem kartesischen Raum. Für die Umsetzung wird im Kern meist ein 3D-Standard wie `OpenGL` genutzt, wobei die tatsächlichen Anzeigeelemente sich in den meisten Fällen nicht von denen bekannter GUIs unterscheidet. Effekte wie Transparenz und Rotation werden ebenfalls verstärkt eingesetzt, die Brauchbarkeit ist allerdings noch Gegenstand von Untersuchungen, und auch für die Generierung von Formularen sind sie vernachlässigbar.

Derzeitige Techniken

Zeitgemäße Oberflächen auf dem Desktop sind sowohl unabhängig vom Betriebssystem als auch daran gebunden verfügbar. Es kommen dabei GUIs zum Einsatz, wobei zumindest auf Desktopebene und zunehmend auch in den Anwendungen ZUI-Ansätze wie skalierbare Icons erkennbar sind.

Die Systeme weisen in der Regel Multitasking-Unterstützung auf, indem jede separat laufende Anwendung als eine Menge von Fenstern dargestellt wird, und man jederzeit zwischen diesen wechseln kann. Die Fenster können dabei Dialoge (und als Teilmenge davon Formulare) sein, es sind aber auch Spezialfenster wie globale Menüs oder nichtmodale *Gadgets* wie etwa virtuelle Notizzettel vertreten.

Realisiert werden diese Anwendungen durch so genannte *Toolkit*-Bibliotheken, von denen meist eine einzige dominant ist und so das Gesamtaussehen eines Desktops bestimmt. Dabei gelten auch bestimmte Konventionen bezüglich des Verhaltens und einiger grafischer Detailfragen, die automatisch generierte Oberflächen unbedingt einhalten müssen, um nicht als „Fremdkörper" wahrgenommen zu werden. Derartige Usability-Betrachtungen werden später noch einmal aufgegriffen werden.

Eine verlässliche Aussage über die Verbreitung der Toolkits und darauf basierender Desktopumgebungen gibt es nicht. Es kann aber angenommen werden, dass es sowohl heute als auch in naher Zukunft bei einem Nebeneinander konkurrierender Ansätze bleibt. Die Generierung von GUIs sollte demnach keine speziellen Toolkits erfordern, sondern möglichst generisch erfolgen, um den größtmöglichen Nutzerkreis erschließend zu können.

2.1.2. Webbrowser

Das Internet hat mit dem *World Wide Web (WWW, oder kurz: Web)* einen Dienst hervorgebracht, der miteinander verknüpfte Dokumente abrufen lässt, welche von den beteiligten Rechnern über das HTTP-Protokoll bereitgestellt werden.

Seit der Mitte der 90er Jahre ist es üblich, WWW-Dokumente nicht nur als Möglichkeit der Präsentation zu betrachten, sondern auch als Medium für die Entwicklung von betriebssystemübergreifenden Anwendungen. Als Seitenbeschreibungssprache ist dabei HTML allgemein anerkannt worden. HTML ist ein Beschreibungsformat für Dokumente, welches semantische Auszeichnungen kennt, diese lassen sich aber mit expliziten Formatierungshinweisen mischen. Derartige Dokumente, auch als Webseiten bezeichnet, sollten sich mit den dafür geschaffenen Anwendungen, den Webbrowsern, ohne Probleme anzeigen und nutzen lassen.

In der Praxis ergeben sich durch oftmals nicht eindeutige oder nicht den Standards entsprechende Webseiten diverse Probleme, die durch teils fehlerhafte Interpretation durch die Browser noch verstärkt werden. Tendenziell tritt aber eine verstärkte Nutzung modernerer XML-basierter Standards wie beispielsweise XHTML [60] auf, deren Verbreitung sich an die von entsprechend neueren Browsern zu koppeln scheint, anstatt anders herum die Verbreitung zu beschleunigen [30][14].

Im Bereich der Formulare, die eine Interaktion mit webbasierten Anwendungen über die Nutzung von Hyperlinks hinaus ermöglichen, vollzieht sich dabei eine parallele Entwick-

lung: Die gebräuchlichen HTML-Formulare (HTML-Forms) werden durch den Standard XForms [11] abgelöst, welcher ab XHTML in der Version 2.0 obligatorisch sein wird.

Insgesamt betrachtet haben sich durch die erweiterten Spielräume bei der Gestaltung von Webseiten eine Menge von Webanwendungen neben traditionellen Desktopanwendungen etabliert. Die deklarativen Angaben der Oberfläche stehen dabei im Vordergrund, aber auch Skriptsprachen wie JavaScript kommen zum Einsatz. Desweiteren werden die Webseiten meist serverseitig in verschiedenen Programmiersprachen erzeugt, was aber für den Rahmen dieser Arbeit ignoriert werden kann.

2.1.3. Mobile Geräte

Neben den klassischen Allzweck-Rechnern haben sich mittlerweile eine Vielzahl von speziellen Geräten und Geräteklassen etabliert, die sich allesamt durch eine geringe Größe und eine gewisse Abgeschlossenheit bezüglich der installierten Software auszeichnen. Es sind dies mobile Geräte, die in der Form von Mobiltelefonen, PDAs und Handhelds auftreten, und dabei spezielle Anforderungen an die Ein- und Ausgabe von Daten stellen. Kleine Displays und Miniaturtastaturen fordern neue Paradigmen auch für die Entwicklung von Bedienoberflächen [34].

Anstelle von vollständigen Dialogen werden beispielsweise oft nur Ausschnitte davon nacheinander angezeigt, ein Prinzip, welches sich Paginierung nennt. Die Navigation erfolgt oft über die Tastatur, im Gegensatz zum Desktop, der Maus und Tastatur gleichermaßen Einsatz gewährt. Es ist anzunehmen, dass über die mobilen Geräte und deren wenig praktikable Eingabemechanismen sich die Sprachsteuerung von Anwendungen verbreiten wird. Für die Unterstützung mobiler Geräte ist es also notwendig, Benutzerschnittstellen erzeugen zu können, die einen multimodalen Zugriff erlauben. Diese Anforderung wird in die weiteren Untersuchungen mit einbezogen werden.

2.1.4. Dialogführung

Unabhängig von der eingesetzten Plattform unterliegt jede Kommunikation zwischen einem Anwender und der Anwendung gewissen Paradigmen. Formular-basierte Anwendungen werden dabei nach einem Dialogprinzip entworfen. Gemäß der DIN-Norm 66234 führt dies zu einem Ablauf, bei dem der Anwender in einem oder mehreren Schritten seine Daten eingibt und anschließend das Ergebnis oder eine sonstige Rückmeldung erhält.

Dialogformen sind unter anderem Kommandosprachen, Menüauswahlen, Masken und Formulare sowie diverse Formen der direkten Manipulation von Objekten. Diese Formen bilden in gewisser Weise die vorgestellten Desktopoberflächentypen TUI, GUI und ZUI ab. Das Aussehen der Oberfläche ist dabei weniger relevant als ihr Verhalten und ihre Zweckmäßigkeit. Eine Auswertung von Experimenten zur Mensch-Maschine-Kommunikation führte dann auch zu folgendem Ergebnis [24].

> Unter dem Aspekt der mentalen Belastung ließen sich für eine direkt-manipulative Benutzungsschnittstelle mit Menüauswahl, Piktogrammen und Maus beim routinemäßigen Bearbeiten einer Graphikaufgabe bei jeweils gleicher

> Leistung keine Vorteile gegenüber einer Interaktion mit abstrakten Kommandokodes feststellen. Die Aufgabenbearbeitung dagegen dauerte mit der direktmanipulativen Interaktionsform notwendigerweise fast zweimal so lange.

Eine Bedienbarkeit einer Anwendung wie des WSGUI-Editors, aber auch der mit seiner Hilfe erzeugten Dialoge, sollte demnach auf alle Fälle mit der Tastatur zusätzlich zur Maus möglich sein. Dies ist meist durch Navigationselemente in einer GUI gegeben. Die Verwendung einer TUI erscheint nicht mehr zeitgemäß, und ZUIs haben, bei all ihren Vorzügen, bisher noch keine weite Verbreitung gefunden.

Dialogformen können hierarchisch oder vernetzt aufgebaut sein, oder aber so, dass der Nutzer nicht viel Spielraum bei der Eingabesequenz hat. Als Dialogform für die Nutzung von Webservices bietet sich eine sequentielle Dialogform bezüglich der Navigation, also Auswahl des Dienstes und der Operation, an. Die Dateneingabe hingegen sollte hierarchisch erfolgen, da auch das Nachrichtenformat derart aufgebaut ist und bei sequentieller Eingabe von größeren Nachrichten ansonsten der Überblick verloren geht.

Weitere Richtlinien wie die DIN EN ISO 9241 (*Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten*) existieren und vertiefen die Betrachtungen zur Ergonomie und Usability [57]. Für den WSGUI-Editor und die mit Hilfe seiner Ausgabe erzeugten GUIs sollen diese Richtlinien intuitiv gelten, ohne dass in dieser Arbeit auf tatsächliche Übereinstimmung überprüft wird.

2.2. Beschreibungsformate

Nach der Betrachtung der Kategorien von Oberflächen wird dieser Abschnitt Sprachen vorstellen, mit denen sich Formulare, Dialoge und Anwendungsfenster beschreiben lassen. Eine Beschränkung auf deklarativ beschriebene GUIs ist im Hinblick auf dynamische Erzeugung von Benutzerschnittstellen für Webservices notwendig und ausreichend zugleich. Die Notwendigkeit ergibt sich dabei aus der Zielstellung der Kombination von Validität (Auswahl eines zum Datentyp passenden GUI-Elements) und Konsistenz (Auswahl der gleichen GUI-Elemente für gleiche Datentypen) [5]. Ausreichend sind die deklarativen Angaben deshalb, weil die Interaktion mit Webservices wenigen vorgegebenen Mustern unterliegt, wie noch gezeigt werden wird.

Dennoch gibt es trotz dieser Beschränkung ein weites Feld an möglichen Dateiformaten mit deklarativen Beschreibungen, die als Ausgangspunkt für die GUI-Erzeugung genutzt werden können. Unterschiede lassen sich im Grad der Implementierung und deren Verfügbarkeit sowie in der Terminologie feststellen. Das Wort GUI kann dabei auf ein vollständiges Anwendungsfenster mit Rahmen, Titelleiste und anderen Elementen bezogen werden, es ließe sich aber auch für ein in einem Fenster befindliches Formular verwenden. Die Abbildung 2.1 erläutert die Begriffe und ihre Relationen zueinander.

Die nachfolgenden Betrachtungen und Evaluationen von GUI-Beschreibungsformaten beziehen sich nun auf Formate, die weitläufig implementiert sind, und teilt diese nach ihrem Umfang bezüglich des Begriffs GUI ein. Es wird dazu in abstrakte, plattformintegrierte und toolkitabhängige Formate unterschieden.

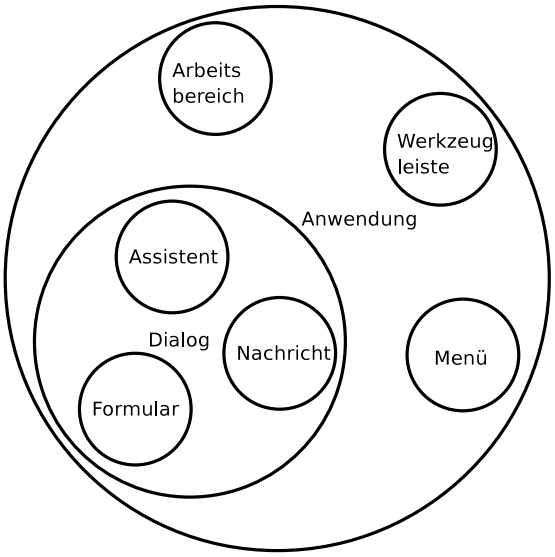

Abbildung 2.1.: GUI-Struktur einer Anwendung

2.2.1. Abstrakte GUI-Beschreibungen

Die erste Gruppe von Formaten bilden solche, die eine abstrakte Beschreibung von GUI-Elementen aufweisen, die je nach Zielplattform unterschiedlich dargestellt werden können. Die gesamte Interaktion mit dem Nutzer ist dabei auf diejenigen Aktionen beschränkt, die sich in dem Beschreibungsformat abbilden lassen. GUI-Elemente sind dabei Knöpfe, Eingabefelder, Auswahlfelder und andere derartige *widgets* [3]. Es können aber darunter auch Containerelemente fallen, die eine Menge von Widgets als Unterelemente enthalten, oder andere grafische Elemente zur visuellen Aufteilung einer Bedienoberfläche.

XForms

XForms [11] definiert Formulare zur Eingabe von Werten an einen Prozess. Dabei kann XForms nie allein eingesetzt werden, sondern benötigt ein umschließendes Dokument, welches die Anordnung der Formularelemente vorgibt. Diese Elemente werden von XForms in abstrakter Form vorgegeben. Je nach umschließendem Host-Dokumenttyp wie XHTML oder SVG können die Elemente dann als GUI- oder ZUI-Elemente dargestellt werden.

Es existiert eine weitgehende Übereinstimmung mit den Elementen von HTML-Formularen, was nicht verwunderlich ist, da XForms der Nachfolger dieser im Nachhinein als *Web Forms* bezeichneten Formulare ist und als abgegrenzter Standard Einzug in den HTML-Nachfolgestandard XHTML 2.0 finden soll [61]. Die Liste der Gemeinsamkeiten beschränkt sich allerdings auf diese Äußerlichkeiten. Intern basieren XForms-Formulare auf einer XML-Instanz [13], welche optional mit Hilfe von XML Schema getypt und strukturiert werden kann. Die Instanz wird initial vorgegeben und im Verlauf der Formularbearbeitung mit Werten aufgefüllt und eventuell strukturell verändert, soweit die

neue Instanz weiterhin dem Schema entspricht. Auch die Übermittlung der Eingaben an den Prozess setzt auf das XML-Format. Dabei läuft die Kommunikation über HTTP-POST, wobei zusätzlich ab Version 1.1 die Möglichkeit besteht, SOAP einzusetzen, um Webservices direkt zu kontaktieren.

Für die Anwendung in einem Webservice-Kontext mit serverseitiger GUI-Generierung ist der SOAP-Transport keine sinnvolle Erweiterung, da ohnehin eine dialog-generierende Software als Instanz zwischen dem Anwender und dem Dienst laufen muss, um so beispielsweise die Ergebnisse anzeigegerecht formatieren zu können.

Die Hauptgründe für den Einsatz von XForms neben der XML-basierten Formularübermittlung sind die Erübrigung clientseitiger Validierungsroutinen etwa in JavaScript, die Möglichkeit mehrerer separater Formulare auf einer Webseite beispielsweise zur Integration in Portale, und der Einsatz moderner Eingabemethoden wie einer Sprachsteuerung für mobile Geräte. Es gibt aber auch Gründe, die dagegen sprechen [35], und die zu einem Alternativvorschlag namens *Web Forms 2.0* führten.

XForms-Implementierungen, sogenannte Prozessoren, existieren für wenige Spezialanwendungen sowie als Erweiterung für Webbrowser der Mozilla-Familie. Aber auch an portablen und erweiterbaren XForms-Prozessoren wird gearbeitet [36], so dass damit zu rechnen ist, in Zukunft eine hohe Bandbreite an XForms-nutzenden Anwendungen finden zu können.

UIML

Die *UI Markup Language (UIML)* [79] ist eine sehr stark abstrahierte Sprache, welche GUI-Elemente nicht anhand ihrer Metaphern wie `window` oder `button` beschreibt, sondern anhand der durch sie repräsentierten Merkmale. Jede UIML-Datei ist dazu in die vier Sektionen Struktur, Stil, Inhalt und Verhalten aufgeteilt [51], aus denen eine Implementierung per Transformation die Domänen Logik und Präsentation erschließen kann.

Da sich UIML nicht mit Layout-Fragen befasst, diese in ersten Versuchen aber als wichtig befunden worden sind, existiert nunmehr die Erweiterung *DI-UIML (Device Independent UIML)*. Dabei werden Positionierungen stets relativ zu Nachbarelementen angegeben. Durch einen Lösungsalgorithmus (*constraint solver*) für eine Menge an Positionierungsdaten kann dann ein passendes Layout berechnet werden.

Implementierungen für UIML gibt es für ein paar Plattformen, etwa für .NET, den Status als wirklich universell verwendete UI-Beschreibungssprache kann man UIML jedoch noch nicht anerkennen. Für die Zukunft liegt an dieser Stelle natürlich noch viel Potential verborgen, da es intuitiv betrachtet für jedes GUI-Toolkit möglich sein muss, durch eine Transformation von UIML-ähnlichen Sprachen in eine toolkit-spezifische Sprache eine Oberfläche generieren zu können.

2.2.2. Plattform-integrierte Anwendungen

Die zweite Gruppe von Beschreibungsformaten umfasst solche, die an und für sich immer noch abstrakt gehalten sind, aber nicht ohne zusätzliche Anbindungen von Komponenten oder Programmiersprachen nutzbar sind. Oftmals werden diese Komponenten durch

eine Laufzeitumgebung, auch Plattform genannt, bereitgestellt. Dabei ist die Laufzeitplattform von der zugrundeliegenden Betriebssystemplattform zu unterscheiden, da insbesondere Portabilität für die Laufzeitplattformen oft ein wichtiges Kriterium darstellt und somit auch ein Einsatz darauf basierender Oberflächenbeschreibungen grundsätzlich überall möglich ist.

XUL

XUL (*XML User Interface Language*) ist aus der Notwendigkeit heraus entstanden, für den Webbrowser Mozilla und seine Abkömmlinge eine GUI-Beschreibungssprache zu haben, welche plattformübergreifend und unabhängig von der Implementierungssprache interpretiert werden kann.

Brauchbare XUL-Anwendungen sind nicht ohne den Einsatz einer Programmiersprache zu erstellen. Dabei wird JavaScript eingesetzt, eine dynamisch getypte und zumeist interpretierte Sprache aus dem Webbereich. Mit Hilfe der Anbindung an den DOM (*Document Object Model*), einer standardisierten Zugriffsmöglichkeit auf Strukturen von XML-Dokumenten, ist es mit JavaScript möglich, bereits angezeigte XUL-Oberflächen zur Laufzeit zu modifizieren. Die Darstellungsvarianten von XUL werden über *cascading style sheets* (CSS) bestimmt. Ähnlich wie auf Webseiten können dadurch die Farbgebung, Platzierung, Schriftarten und Sichtbarkeit von Elementen eingestellt werden.

Neben XUL, JavaScript und CSS sei an dieser Stelle noch auf XPCOM hingewiesen, ein Komponentenmodell zur Anbindung nativer Prozeduren an XUL, sowie auf XBL, eine Sprache zur deklarativen Gestaltung von neuen GUI-Elementen. All diese Bestandteile zusammen ergeben die XPFE-Architektur (*Cross Platform Front End*), welche durch die Mozilla-Implementierung Gecko auf eine GUI abgebildet wird [40]. Reine XUL-Oberflächen ohne Unterstützung der XPFE sind demnach selten anzutreffen und können auch nicht das mögliche Potential ausreizen.

XAML

Nachdem XUL-Anwendungen recht erfolgreich entwickelt worden sind, wurde mit XAML (*eXtensible Application Markup Language*) ein ähnlicher Ansatz vorgestellt, der jedoch als Plattform nicht die Mozilla-Browserfamilie, sondern *Microsoft Windows* benötigt und somit nur bedingt plattformübergreifend arbeiten kann. Die geplante Verschmelzung des betriebssystemunabhängigen Web mit Desktopanwendungen [30] wird dadurch wohl nicht erreicht werden. Aufgrund der sonstigen Gemeinsamkeiten mit XUL und einer noch andauernden Entwicklung inklusive dem Mangel einer Spezifikation soll das Format an dieser Stelle nur der Vollständigkeit halber erwähnt sein.

2.2.3. Toolkit-spezifische Formate

Die dritte und letzte Gruppe von Beschreibungsformaten umfasst jene, die speziell für eine konkrete Bibliothek entwickelt worden sind - oft erst im Nachhinein, um beispielsweise ein dynamisches Nachladen von Dialogen in einer Anwendung zu ermöglichen. In anderen

Fällen ist eine Umwandlung der Beschreibung in Quelltext als statische Konvertierung notwendig.

Qt UI

Die Bibliothek Qt bringt ein eigenes UI-Beschreibungsformat mit, welches nicht formal unter einem Namen spezifiziert ist und gemeinhin als *Qt Designer UI* bezeichnet wird, benannt nach dem Programm Designer, welches die Erstellung dieser `ui`-Dateien erlaubt.

Das XML-basierte Format dient der Beschreibung von Anwendungsfenstern, also entweder einem Hauptfenster mit Menüleiste, Werkzeugleiste und Statusanzeige oder einem Dialogfenster in verschiedenen Varianten. Diese Fenster werden dabei als *form* (Formular) bezeichnet, die Beschreibung ist aber nicht auf Formulare beschränkt oder optimiert.

Als Elemente stehen einfache und komplexere Widgets zur Verfügung, so beispielsweise Listen-, Tabellen- und Baumansichten. Die einfachen Widgets werden explizit in solche zur Eingabe (Texteingabe, Auswahlfeld, Scrollleiste) und zur Ausgabe (Markierung, LCD-Anzeige, Fortschrittsbalken) eingeteilt. Als weitere Elemente stehen noch horizontale und vertikale Trennlinien mit wahlweiser Positionsverschiebung sowie sichtbare Containerelemente (Werkzeugbereich, Widget-Stapel, Rahmen, andockbare Subfenster) zur Auswahl bereit. Die Menge an verfügbaren Widgets für das Beschreibungsformat kann dabei über die Grundmenge hinaus durch weitere Widgets erfolgen, welche einem bestimmten Format genügen und nach ihrer Installation durch Qt Designer auffindbar sind.

Layoutinformationen und pixelgenaue Positionierung sind in diesem Format möglich. Auch Metainformationen über den Autor und Kommentare werden unterstützt. Dialogdynamik wird über das Qt-Konzept von Signalen und Slots (Signalempfängern) erreicht: wird auf einem Widget ein Ereignis ausgelöst, so kann man dieses an andere Widgets oder der Dialog selbst weiterleiten. So kann beispielsweise ein Drehregler mit einer Anzeige versehen sein, die stets den aktuellen Wert des Reglers anzeigt. Signal und Empfänger müssen dabei die gleiche Typsignatur haben, die sich aus einer Anzahl von Parametern zusammensetzt, welche jeweils typgleich sein müssen. Flexibilität wird dadurch erreicht, dass Slots dabei einige der Parameter verwerfen können.

Sinnvoll ist der Mechanismus erst dann, wenn eigene Signalempfänger bereitgestellt werden können. Dies ist jedoch nicht deklarativ möglich. Der Hauptanwendungszweck der Dialoge ist es also, mit Hilfe des UI-Compilers `uic` Quelltext in Form einer C++-Klasse erzeugen zu lassen, und von dieser Klasse ableitend die Anwendungslogik zu implementieren.

Dialoge, die einem bestimmten Muster folgend erzeugt werden, können aber durch ihre einbindende Anwendung dynamisch mit Signalverbindungen versehen werden. Auch das Auslesen der Datenfelder ist über eine spezielle API auf generische Art und Weise möglich.

Glade

Die Widgetbibliothek Gtk+ definiert ebenfalls ein eigenes XML-basiertes Format zur Beschreibung der mit ihr erstellten Oberflächen. Dieses Format wird nach dem ursprünglichen Designwerkzeug dafür als Glade bezeichnet. Es unterscheidet sich nicht wesentlich von Qt UI, und soll deshalb nicht vertiefend dargestellt werden. Der einzige nennenswerte Unterschied besteht darin, dass in Glade zwar Signale, aber keine Empfänger deklarativ angegeben werden können, und somit eine dialoginterne Dynamik verhindert wird.

XSWT

Aus dem Java-Bereich entstammt XSWT, eine deklarative Beschreibung von Anordnungen diverser GUI-Elemente, die dem Toolkit SWT (*Standard Widget Toolkit*) entstammen. Dieses Toolkit gilt als Alternative zu den Java-Standardtoolkits AWT und Swing und ist durch die Eclipse-Plattform populär geworden. Als Besonderheit gegenüber den anderen Toolkits ist SWT im Grunde genommen nur eine Abbildung auf native Toolkits der jeweiligen Plattform, um zum einen die Leistungsfähigkeit zu erhöhen, zum anderen aber auch die Akzeptanz beim Anwender zu steigern. So passen sich SWT-Anwendungen stets dem aktuellen Desktop an. Während Qt diese Aufgabe durch unterschiedliche Implementierungszweige selbst übernimmt, stützt sich SWT auf bereits vorhandene Toolkit-Bibliotheken. Ein weiterer Unterschied existiert in der Ausdrucksfähigkeit des Beschreibungsformats. GUI-Ereignisse können in Qt UI durch Signale und Slots direkt an andere GUI-Elemente geknüpft werden, während XSWT dazu Anweisungen als Java-Code benötigt. Ein relevanter Anwendungsfall ist die Änderung einer Anzeige bei Betätigung eines Schiebereglers (*slider*) [53].

Eine XSWT-Alternative namens Reflexive Interface Builder (RIB) soll Verbesserungen bringen. Im Unterschied zu XSWT werden dabei sowohl SWT als auch AWT/Swing unterstützt, was auf eine höhere Abstraktionsstufe schließen lässt. RIB ist dabei sowohl die Bezeichnung für das Dateiformat zur Beschreibung der Oberflächen als auch der Name für das Programm zur Anzeige derartiger Dateien [27]. Das RIB-Dateiformat ist dabei an Java gebunden und nutzt dessen Möglichkeiten zur Objektselbstprüfung (*introspection*) aus, um Verknüpfungen zwischen den definierten GUI-Elementen bei Auslösung einer Aktion herzustellen. Dieser Ansatz behebt das Problem der notwendigen Code-Anbindung wie in XSWT, entspricht aber gleichzeitig nicht zu einer vollständig deklarativen Angabe wie in XForms. Stattdessen wird Java-Code abschnittsweise in RIB verwendet. Alternativ stehen Anbindungen an Scriptsprachen wie Ruby und Python zur Verfügung. Damit verbunden ist eine Verlagerung der Abhängigkeiten von der Anwendung in das Format, jedoch keine Reduktion derselben.

Als Ergänzung zu diesem Format sei noch gesagt, dass genau genommen sämtlicher Java-Code serialisiert und damit als Datenformat abgelegt werden kann. Dies wird zwar auch in der Praxis vollzogen [49], entspricht aber weiterhin einer nicht-deklarativen Angabe, und ist zudem noch abhängig von der eingesetzten Java-Version.

Format	Logikelemente	Imperative Elemente
XForms	Berechnungen und Aktionen	nein
UIML	nein	nein
XUL	nein	JavaScript
XAML	unbekannt	unbekannt
Qt UI	Signale und Slots	nein
Glade	nur Signale	nein
XSWT	nein	nein
RIB	Beschränkungen	Java/Python/Ruby

Tabelle 2.1.: GUI-Formatbeschreibungen

2.3. Funktionalität

Eine wichtige Frage nach der Erzeugung einer Oberfläche lautet, wie ihre Interaktion mit dem Benutzer ablaufen soll. Die vorgestellten deklarativen Beschreibungsformate enthielten alle eine unterschiedliche Menge an logischen Verknüpfungen und imperativen Bestandteilen. Die Logikelemente sind dabei innerhalb des Formats spezifizierte Abhängigkeiten und Aktionen sowie eine mögliche Anbindung einer Anwendung, während die imperativen Elemente eine freie Programmierbarkeit zulassen. Der Kompaktheit halber sind diese Unterschiede noch einmal in Tabelle 2.1 zusammengefasst.

Formate mit imperativen Sprachelementen bieten diese nicht nur an, sondern erfordern sie oftmals auch für die sinnvolle Nutzung der über das Format erzeugten Oberflächen. Die vollständige Deklarativität von XForms schlägt sich in einem komplexen und schwer verständlichen Standard nieder, ist jedoch von Vorteil für die automatische Generierung einer nicht nur ansehnlichen, sondern auch funktionalen Oberfläche.

2.4. Bewertung

Die Vielzahl an Dateiformaten zur GUI-Generierung suggerieren, dass diese Möglichkeit auch ausgeschöpft wird. Tatsächlich steigt die Zahl der Bedienoberflächen, die zumindest teilweise generiert sind, an [62]. Fachlich gesehen ist GUI-Generierung aus einem Modell neben der Code-Generierung ein wichtiger Aspekt der modellgetriebenen Entwicklung (MDA) [29], und somit kommt auch die Unterscheidung zwischen plattformübergreifender (abstrakter) und plattformspezifischer GUI-Beschreibung zum Tragen [67].

Die Abbildung 2.2 demonstriert, wie über verschiedene Schritte vom Modell ausgehend über die Beschreibungsformate die GUI auf einer Plattform erzeugt wird.

Die Verwendung eines toolkit-spezifischen Formats ist dabei eine Möglichkeit. Dieser in Schritt (1) beschrieben Weg wird oft von sogenannten GUI-Buildern wie den vorgestellten Werkzeugen Qt Designer und Glade vorgenommen. Wird hingegen eine abstrakte Beschreibung gewünscht, so kann Schritt (2) durch einen GUI-Builder oder durch Umwandlung des Datenmodells begangen werden. Genau dieser Schritt wird die Domäne

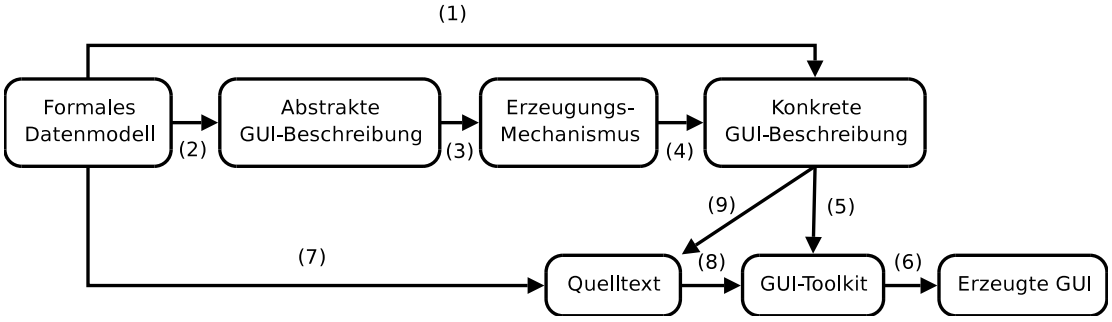

Abbildung 2.2.: Generierungsprozess vom Modell zur GUI

des WSGUI-Editors sein. In diesem Fall folgt in Schritt (3) und (4) die Anpassung auf die jeweilige GUI-Bibliothek und auf die Zielplattform. Beide Wege führen dann über das GUI-Toolkit (5) hin zu einer erzeugten GUI (6). Der traditionelle Weg zur manuellen GUI-Erzeugung läuft über die Erstellung eines Quelltextes durch den Entwickler (7) und Aufrufen der GUI-API im Quelltext (8). Es ist auch möglich, den Quelltext aus der konkreten Beschreibung zu erzeugen, wie in Schritt (9) gezeigt. Schritt (8) folgt analog der vorherigen Methode.

Für die Verwendung in einer Routine zur GUI-Generierung ist es nicht praktikabel, zusätzlich zu einer GUI-Beschreibung noch Quelltext in einer bestimmten Programmiersprache einsetzen zu müssen. Eine Überführung eines XML-Quellformates in ein XML-Zielformat, welches eine GUI beschreibt, ist bei rein deklarativen Angaben flexibler lösbar, nämlich neben programmatischen Transformationen auch über XML-Transformation mit Stylesheets (XSLT) [17]. Eine Abschwächung dieser Aussage liefern interpretierte Sprachen wie JavaScript, was sich jedoch negativ auf die Anforderungen an die Laufzeitumgebung niederschlägt, da ein Sprachinterpreter stets Ressourcennutzung wie Rechenzeit und Speicherbedarf ansteigen lässt. Die Verwendung von XForms für die Beschreibung von GUI-Elementen ist für formularzentrische Dialogführung deshalb nur konsequent. Die Umsetzung des Model-View-Controller-Prinzips (MVC) trennt sauber das in XML Schema angegebene Objekt von der erzeugten GUI. Auch der multimodale Zugriff für mobile Geräte wird durch XForms begünstigt [71].

3. Einführung in WSGUI

Die Ausarbeitung der gewünschten Eigenschaften von Oberflächen und daraus resultierend die Auswahl an Formaten für die Erzeugung von GUIs aus Webservices stellt hohe Ansprüche an die Algorithmen zur Generierung. Es sollen dafür die gegebenen Informationen in Webservice-Beschreibungen untersucht werden mit der Zielstellung, diese soweit wie möglich zu nutzen und überall dort, wo dies nicht möglich ist oder Angaben fehlen, Zusatzinformationen bereitzustellen. Der WSGUI-Editor wird also die Angabe von zusätzlichen Informationen in Abstimmung mit den zu generierenden Dialogen ermöglichen müssen.

In diesem Kapitel werden folglich zuerst die existierenden Schnittstellenbeschreibungen für Webservices und die Funktionsweise einer bestimmten Software zur Generierung von GUIs im Rahmen der Interaktion mit Webservices erklärt. Die Software unter dem Namen *WSGUI-Engine* und ihre Merkmale sind Teil der Erklärungen. Die durch sie umgesetzten WSGUI-Konzepte und alternative Ansätze werden erläutert. Anschließend werden die Webservice-Schnittstellenbeschreibungen erneut herangezogen, dann allerdings vor dem konkreten Hintergrund einer Umwandlung in XForms durch die WSGUI-Engine.

3.1. Beschreibung von Diensten

Das Dateiformat WSDL (*Web Service Definition Language*) ist das gängige und bekannteste Format zur Beschreibung von Diensten, ihren Operationen und deren Nachrichten, sowie dem Aufbau von Nachrichten. Die aus der CORBA-Welt bekannten IDL-Dateien sind ansatzweise noch wiederzuerkennen, allerdings beschreiben WSDL-Dateien eben Dienste (genauer gesagt Webservices) und keine Komponenten [16].

Die Illustration 3.1 zeigt stark vereinfacht den Aufbau einer WSDL-Datei und den strukturellen Weg vom Dienst zur Nachricht, welcher durch sie beschrieben wird.

Neben der Struktur eines Dienstes ist auch dessen Anbindung an die Außenwelt beschrieben. WSDL-Dateien beinhalten dazu Bindungen zu HTTP- und SOAP-Endpunkten.

Das *Simple Object Access Protocol* (SOAP) [55] dient dabei dem Transport von Nachrichten, welcher in der WSDL-Datei definiert sind. Diese werden mit ein paar Zusatzinformationen versehen und dann über ein Basisprotokoll wie HTTP übertragen, woraufhin Antwortnachrichten empfangen werden können. Neben den Nachrichten können noch Binärdaten quasi als Anhänge mit jeweils einem spezifischen MIME-Typ Teil einer SOAP-Nachricht sein. SOAP-Nachrichten sind ansatzweise mit *remote procedure calls*, also RPC-Nachrichten, vergleichbar. Ihre Angabe in den WSDL-Dateien beinhaltet auch ähnliche Aufrufmuster: Der Aufrufer kann einen Webservice kontaktieren und die Antwort dazu erhalten, die Interaktion kann jedoch auch anders herum geschehen, und die Antwort kann auch ausbleiben.

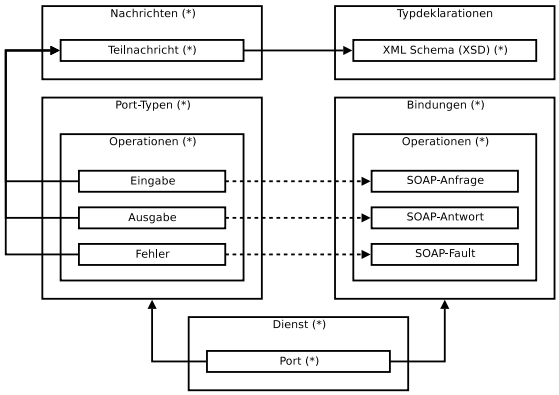

Abbildung 3.1.: Aufbau einer WSDL-Datei

Der Aufbau von ausgetauschten Nachrichten wird über XML Schema (XSD) beschrieben [26]. Die entsprechenden XSD-Abschnitte befinden sich dabei innerhalb einer WSDL-Datei, können jedoch zumindest teilweise über `include`- und `import`-Anweisungen aus weiteren Dateien importiert werden.

Neben WSDL-Dateien mit integrierten XSD-Abschnitten gibt es auch weitere Beschreibungsformate für Webservices [20]. Diese werden aber der Kompaktheit halber nicht mit betrachtet.

3.2. Dynamische Aufrufe und Interaktion

WSDL als Beschreibungsformat dient hauptsächlich als Kontrakt zwischen zwei Kommunikationspartnern, etwa einem Dienst und einer aufrufenden Anwendung. Die Anwendung kann somit überprüfen, ob das vom Dienst verlangte Nachrichtenformat eines ist, welches die Anwendung erzeugen kann, ob eine Verschlüsselung der Übertragung notwendig ist und auf welchem Weg die Kommunikation überhaupt stattfinden soll.

Nun gibt es seit der Einführung von WSDL-Dateien Pläne, derartige Aufrufe nicht nur durch Anwendungen, sondern auch durch Menschen tätigen zu lassen. Die Natur der WSDL-Dateien mit ihren Variablennamen, oft fehlender Dokumentation zu den einzelnen Operationen sowie datenzentrischer Beschreibung der Nachrichten führte alsgleich dazu, dass zusätzliche Informationen bereitgestellt wurden, die auf das WSDL-Format angewendet eine brauchbare Basis ergaben.

Vor dem Aufruf einer Operation in einem Webservice trifft der Nutzer zu Beginn eine Dienstauswahl, und eventuell noch einmal eine Operationsauswahl, sofern der Dienst mehrere Operationen anbietet. Anschließend wird das Eingabeformular erzeugt. Der Nutzer gibt seine Daten ein und sendet das Formular ab. Er erhält daraufhin eine Anzeige der Ergebnisse. Im Rahmen komplexer Webservice-Prozesse kann basierend auf der Ausgabe die Eingabe für den nächsten Aufruf erfolgen.

WSDL-Dateien definieren vier verschiedene Aufrufmuster für Webservices: Einwegau-

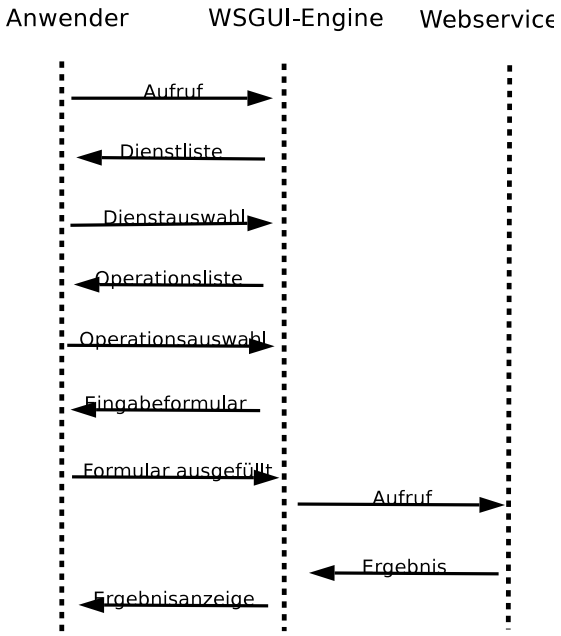

Abbildung 3.2.: Ablauf der Interaktion mit einem Webservice

fruf ohne Antwort (*one-way message*), Anfrage mit Antwort (*request-response message*), Rückrufanfrage (*solicit-response message*) und Benachrichtigung (*notification message*). Dabei sind nur die ersten beiden Muster vom Nutzer initiiert und in der WSDL-Spezifikation mit Bindungen versehen, während die anderen beiden Muster vom Dienst initiiert werden und somit für die weitere Betrachtung ignoriert werden können. Die reine GUI-Erzeugung in WSGUI-Engines ist zwar nachrichtenbasiert und somit spielt das Aufrufmuster dafür keine Rolle, die Interaktion mit dem Nutzer hingegen ist mit dienstinitiierten Aufrufen schwieriger zu realisieren. Man unterteilt deshalb auch in einfache und komplexe Webservices und daraus abgeleitet verschiedene Formen der Dialogführung.

Das Ablaufdiagramm 3.2 illustriert die Interaktion mit einem einfachen Webservice. Ein solcher Ablauf soll als Einsatzszenario für den WSGUI-Editor in dieser Arbeit genügen. Die Erweiterung von WSGUI-Konzepten auf komplexe Dienste wurde bereits an anderer Stelle untersucht [77][9][6] und ließe sich bei Bedarf mit einem modifizierten WSGUI-Editor realisieren.

3.3. Begriffsdefinition WSGUI

Der Begriff WSGUI steht für *Web Services Graphical User Interface* [74] und bezeichnet eine Familie von Konzepten, welche allesamt implizite und explizite Hinweise zur Generierung von graphischen Benutzerschnittstellen ausgehend von normativen Datenmodellen

angeben. Derartige Datenmodelle findet man hauptsächlich in den Typbeschreibungen der Nachrichten von Webservices, wie im vorhergehenden Abschnitt beschrieben.

WSGUI-ähnliche Konzepte gibt es seit etwa 2002, der Begriff WSGUI und die analytische Betrachtung spezieller Eigenheiten der Generierung XForms-basierter GUIs für Webservices gehen zurück auf eine Veröffentlichung aus dem Jahr 2003 [41].

Als Beschreibungsformat für die Beeinflussung der GUI-Generierung existiert das Dateiformat GUIDD (*Graphical User Interface Deployment Descriptor*) [42]. Es enthält Formularkomponenten (`formComponents`) für jedes Element einer Nachricht, welche die GUI-Elemente von XForms, die sogenannten *XForms controls*, referenzieren, potentiell jedoch auch weitere Klassen von GUI-Elementen mit aufnehmen könnten. Speziell für GUI-Elemente für die Ausgabe sind `outputTypes` vorgesehen, die den MIME-Typ der dargestellten Daten angeben. Desweiteren kann mit GUIDD der Dienst und die Operationen beschrieben werden, insbesondere dann, wenn die entsprechenden Felder in der WSDL-Datei nicht genutzt werden oder Übersetzungen hinzugefügt werden sollen. Neben derartigen `service`- und `operations`-Einträgen gibt es auch eine Liste von `stylesheets`, um für die Einbindung in Webseiten eine Anpassbarkeit vornehmen zu können. Das GUIDD-Format ist noch nicht als endgültig anzusehen, hat sich jedoch in praktischen Anwendungen als vom Umfang her hinreichend für die Generierung nutzbarer GUIs zur Bedienung von Webservices gezeigt [75].

Eine WSGUI-Engine bezeichnet dabei die Software, welche als generische Anwendung agiert und die Oberflächen für Ein- und Ausgabe entsprechend des Zustands der Dienstbedienung und abgestimmt auf die Zielplattform des Nutzers generiert. Die Nutzung von WSGUI-Informationen kann dabei durch Einlesen von GUIDD-Dateien und der Abbildung ihres Inhalts auf die jeweilig referenzierten WSDL- und XSD-Elemente stattfinden.

Ein Beispiel für eine WSGUI-Engine ist der vom Autor entwickelte *Dynvoker*. Dieses Servlet wird in einem späteren Kapitel genauer beschrieben werden.

3.4. WSGUI-Konzepte

Die Arbeitsweise einer WSGUI-Engine lässt sich in mehrere Schritte oder Stufen zerlegen. Je nach gewünschter Qualität der erzeugten GUI kann eine beliebige Kombination der Stufen durchlaufen werden, wobei die Reihenfolge vorgegeben ist. Die Abbildung 3.3 zeigt die fünf wichtigsten Stufen. Ein Inferenzmechanismus bildet dabei die Grundlage und sorgt für die Erzeugung von typsicheren Eingabemasken. Alle weiteren Stufen sind nur mit Hilfe zusätzlicher Informationen erreichbar. Die Erzeugung typsicherer Eingabefelder auf funktionale Art und Weise ist auch in [31] erläutert.

Der generalisierte Ablauf entspricht der GUI-Erzeugung in vielen Bereichen der Softwaretechnik, etwa der modellgetriebenen Entwicklung [48]. Im Webservice-Umfeld unter Verwendung von XML Schema als Eingabeformat und XForms als Ausgabeformat lässt sich das generelle Schema verfeinern. Die Nutzung einer GUIDD-Datei in der Implementierung Dynvoker ist in Abbildung 3.4 dargestellt.

Die GUIDD-Datei beinhaltet Formularelemente, welche in der zweiten Generierungsstufe zum Einsatz kommen und eventuell durch Inferenz gewonnene ersetzen. Jedes For-

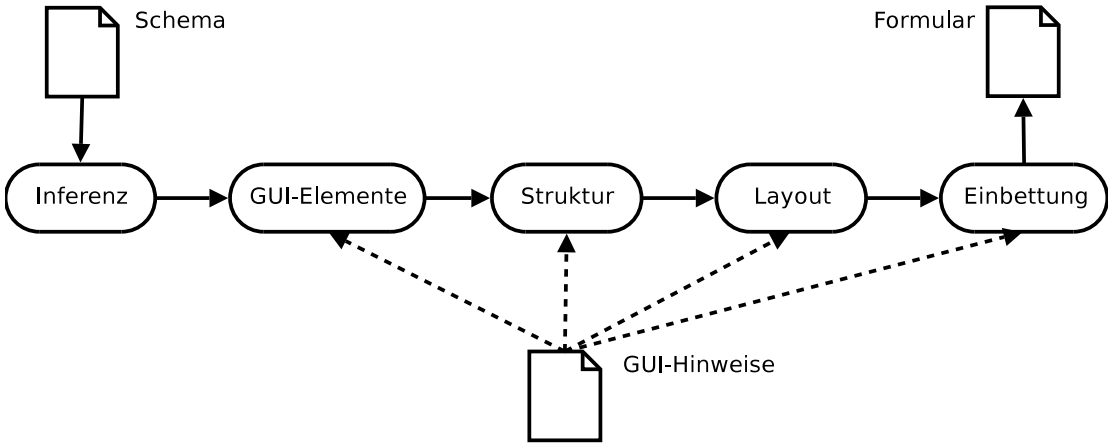

Abbildung 3.3.: Generelle Arbeitsschritte einer WSGUI-Engine

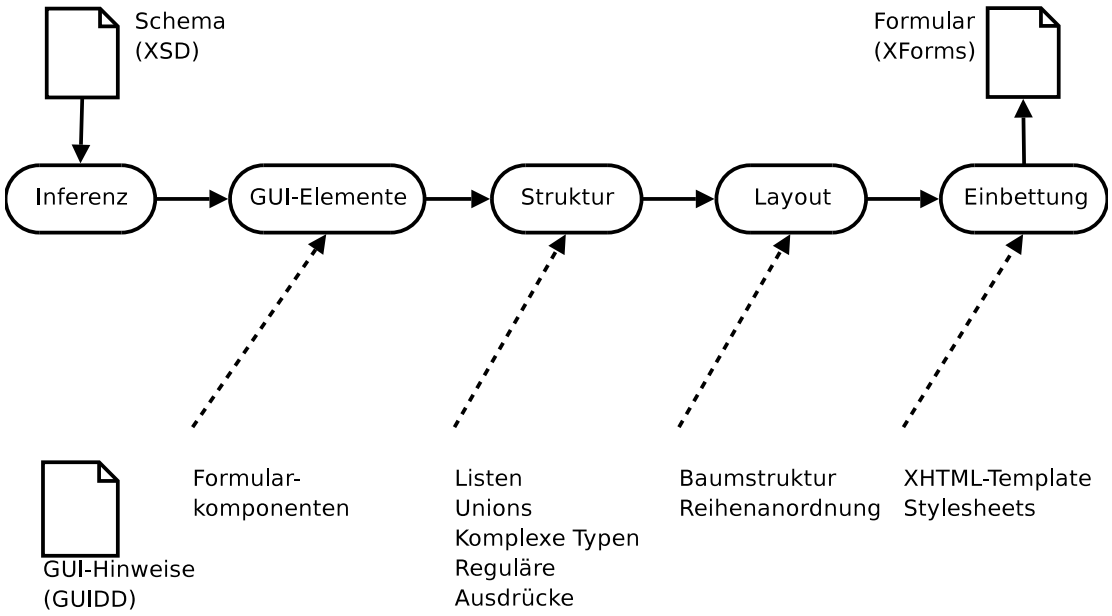

Abbildung 3.4.: Beispielhafte Implementierung im Dynvoker

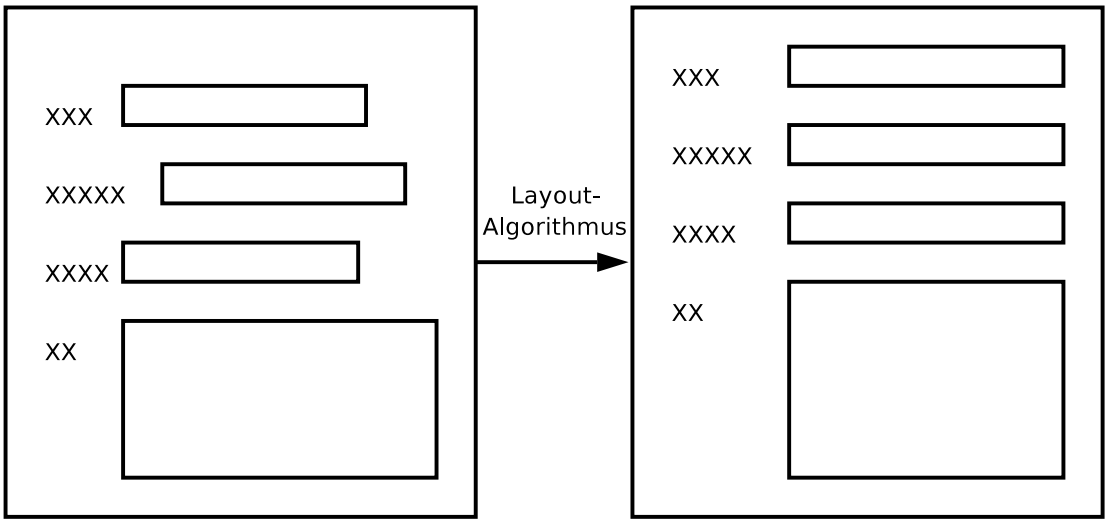

Abbildung 3.5.: Positionierungsstufe der WSGUI-Verarbeitung

mularelement ist dabei ein XForms-GUI-Element, welches eventuell durch Übersetzungen erweitert wird. Die Adressierung der Position im Schema oder der Instanz wird durch den Eintrag von Attributen `xpath` oder `ixpath` in die umschließende Formularkomponente erreicht. Die XForms-Elemente werden dabei über Filter vorverarbeitet, um beispielsweise nur eine der vorhandenen Übersetzungen auszuliefern oder fehlende Tipps und Dokumentation zu den Elementen (`hint` bzw. `help`) durch vorhandene `documentation`-Einträge in WSDL-Dateien zu ersetzen.

Sofern ein Element nicht nur einmal, sondern mehrmals auftreten kann (Liste), oder es nur im gegenseiten Ausschluss mit einem anderen Element auftritt (Union), muss die WSGUI-Engine dafür sorgen, die GUI-Generierung entsprechend anzupassen. Hierin liegt ein Großteil der Generierungslogik. So müssen für Listen variabler Größe Kontrollelemente zum Hinzufügen und Entfernen zusätzlicher Listenelemente eingefügt werden. Unions hingegen benötigen als Kontrollelement einen Auswahlschalter.

Die Positionierung von Elementen wird in der vierten Stufe der WSGUI-Pipeline berechnet. Dabei sind Richtlinien zur Layoutgestaltung zu beachten [48][6]. Das Ergebnis automatisch generierter Layouts kann bei wirksamen Algorithmen eine konsistente Benutzerführung ohne verwirrend platzierte GUI-Elemente sein, wie in Abbildung 3.5 schematisch dargestellt.

Sofern ein Eingabe- oder Ausgabeformular erzeugt wurde, kann dieses beim Einsatz auf einer Webseite noch in eine XHTML-Vorlage eingebunden werden. Auch die zusätzliche Einbindung von Stylesheets ist möglich. Diese fünfte und letzte Stufe schließt den Generierungsvorgang ab, die erzeugte Webseite wird dann an die Clientanwendung ausgeliefert.

Im Kapitel über die Konzeption des Editors wird auf die einzelnen Konzepte nochmals bewertend eingegangen werden. Um verstehen zu können, welche Konzepte für den WSGUI-Editor relevant sind und welche nicht umgesetzt werden können oder brauchen,

soll zuvor eine Reihe von Alternativen und anschließend eine Analyse von WSDL-Dateien präsentiert werden.

3.5. Alternative Ansätze

Es gab und gibt weitere Ansätze neben WSGUI, um grafische Bedienoberflächen für Webservices zu erzeugen. Die Verfahren WSUI, WSXL und WSRP werden als hauptsächliche Vertreter vorgestellt, ergänzt um ein paar Aufrufimplementierungen.

Bereits im Jahr 2001 wurde die Notwendigkeit von zusätzlichen GUI-Informationen aufsetzend auf Webservice-Beschreibungen erkannt. Die *WSUI-Initiative (Web Services User Interface)* stellte dabei einen Entwurf für eine Spezifikation online, der allerdings wie die gesamte Webseite nicht mehr verfügbar ist. Ein Vergleich mit WSGUI beschränkt sich aus diesem Grund auf ein paar veröffentlichte Artikel zu dem Thema [2] mit entsprechendem Mangel an Tiefe.

WSUI wurde hauptsächlich von Portalanbietern vorangetrieben und hatte die Verwendung von HTML zur Annahme. Unklar ist, ob auch andere Formate angedacht waren. Die starke Verwendung von XSL-Transformationen und die Beschreibungen lassen einen solchen Schluss aber nicht zu. Die WSUI-Konzepte schienen sehr auf die Beschreibung von GUI-Elementen fixiert zu sein, von einem Äquivalent zur WSGUI-Engine findet man keine Hinweise in der Literatur.

WSUI scheint dabei der einzige mit WSGUI vergleichbare Ansatz gewesen zu sein, wie die Referenzen in den relevanten Publikationen erkennen lassen. Es gab und gibt hingegen speziell im Portalbereich noch weitere Verfahren, um GUIs dynamisch nutzen zu können. Zwei Vertreter davon sollen dabei kurz besprochen werden: WSXL und WSRP. Dabei sei jedoch gesagt, dass die Daten zu diesen Verfahren ähnlich dem Prinzip der Webseiten serverseitig bereits existieren müssen und nur dynamisch in die GUI des Clients, beispielsweise als Portlet in ein Portal, eingebunden werden.

WSXL, die *Web Services Experience Language*, ist eine im April 2002 vorgestellte Sprache zur Verwaltung und Verteilung von Anwendungen im Webservice-Umfeld [37]. Dabei ist WSXL stark komponentenorientiert aufgebaut. Neben der Möglichkeit einer individuellen Gestaltung existierender Anwendungen (*OEM branding*) und der Überwachung des Lebenszyklus können über spezielle Adapter auch Instanzdaten sowie CSS-Dateien modifiziert und eingefügt werden. Grundsätzliche Änderungen an der GUI sind aber nicht möglich. Die WSXL-Operation `getOutput()` dient der Erfragung einer kanonischen GUI, auf die höchstens besagte Änderungen angewandt werden können.

WSRP (*Web Services for Remote Portlets*) ist noch weiter als WSXL im Bereich Portale angesiedelt und dient einzig und allein der Einbindung von Portlets aus anderen Portalen, die ein paar ihrer Portlets exportieren. Dabei müssen die Portlets dem Standard *JSR-168* entsprechen. Durch WSRP können ähnlich wie bei WSXL Anpassungen an den HTML-Inhalten vorgenommen werden, so etwa eine automatische Änderung von Hyperlinks, damit diese nicht mehr auf das Ursprungsportal verweisen, sondern auf das in der neuen Umgebung.

Implementierungen, die mit dem Dynvoker vergleichbar sind, gibt es trotz des Man-

gels an Alternativen zu definierten WSGUI-Konzepten viele. Meist werden solche Konzepte unwissentlich eingesetzt, manchmal werden die resultierenden GUIs aber auch als brauchbar eingestuft. Auf die Implementierungen soll an dieser Stelle nur soviel eingegangen werden wie nötig ist, um einen Überblick zu bekommen und abschätzen zu können, inwiefern sie als Testanwendungen für die vom WSGUI-Editor erzeugten Dateien dienen können. Als Implementierungen wurden Softwareprojekte mit den vielklingenden Namen Forms Generator, SOAPscope, Kung, Gemstone und Swing-Generator ausgewählt.

Der XML Forms Generator [43] ist ein Plug-in für Eclipse, welches sowohl XML-Instanzdateien visualisieren als auch dynamisch Webservices aufrufen kann. Dabei werden XForms-Anzeigen aus WSDL-Dateien generiert. Es können in einer aktuellen Version auch externe Übersetzungskataloge genutzt werden.

Das SOAPscope ist eine Anwendung, die sich eher an Entwickler richtet. Sie versucht nicht, die XML-Strukturen der Nachrichten zu verstecken, sondern integriert strukturelle Merkmale in die generierte GUI. Auch die SOAP-Nachrichten werden nach jedem Aufruf angezeigt.

Kung ist der Webservice-Aufrufer aus dem Kode-Projekt [69] von KDE [23]. Er ist als Desktopanwendung realisiert und aus diesem Grund in jede andere Anwendung einbindbar. WSGUI-Konzepte werden von Kung kaum unterstützt. Interessant ist jedoch die automatische Untersuchung von Ausgabewerten in den XML-Daten auf ihren MIME-Typ und die Auswahl einer passenden GUI-Komponente wie etwa einem Bildbetrachter bei Vorliegen von Bilddaten.

Gemstone ist eine Oberfläche mit verschiedenen kleineren Anwendungen, äußerlich einem Portal ähnlich, jedoch vollständig auf Aufrufe von Webservices optimiert [47]. Dabei werden die Oberflächenbeschreibungen (sogenannte *Gems*) für jeden Webservice manuell erzeugt und dynamisch in die Oberfläche geladen. Im Gegensatz zum WSGUI-Ansatz findet also keine automatische Generierung der Oberfläche statt, es existiert anders als in herkömmlichen Anwendungen aber sehr wohl die Trennung zwischen Applikationslogik und Oberfläche. Für die Implementierung wird XPFE verwendet, also die deklarative Sprache XUL in Verbindung mit JavaScript für die Interaktion und CSS für die Präsentation.

Ein Prototyp einer dem Dynvoker ähnlichen Implementierung existiert für die Erzeugung von Swing-Dialogen. Dafür wird nicht etwa ein Ansatz wie XSWT genutzt, sondern die Swing-GUI wird als Serialisierung von Java-Objekten erzeugt. Diese etwas ungewöhnliche Art der GUI-Generierung ist zur Zeit aber noch nicht dynamisch implementiert, sondern benötigt Vorlagedateien speziell für Strukturen wie Listen oder Unions, welche manuell hinzugefügt werden müssen [49]. Inferenzkonzepte sind aber ähnlich wie im Dynvoker umgesetzt. Ein Unterschied besteht darin, dass der Prototyp nur auf Schemadateien arbeiten kann und keine Webservice-Aufrufe ermöglicht, während Dynvoker sowohl XSD-Dateien als auch WSDL-Dateien verarbeiten kann.

Weitere Implementierungen sind unter anderem der dynamische Aufrufer in der IDE Eclipse und die automatisch generierten HTML-Schnittstellen zu Webservices basierend auf Apache Axis [21] und der .NET-Plattform. Dazu kommt noch SOAPclient, ein in verwandten Arbeiten oft erwähnter Dienst zur einfachen Webservice-Interaktion mit HTML-Ausgabe der Ergebnisse [78]. All jene Aufrufer haben deutliche Schwierigkeiten

Engine	Plattform	Generierung	WSGUI	WS-Aufruf
Dynvoker	Webseiten (XForms)	ja	ja	ja
Forms Generator	Eclipse (XForms)	ja	teilweise	ja
SOAPscope	Webseiten (HTML)	ja	nein	ja
Kung	Desktop	ja	nein	ja
Gemstone	Webseiten (mit XUL)	nein	nein	ja
Swing-Generator	Desktop	ja	teilweise	nein

Tabelle 3.1.: Generierung von GUIs aus Webservices

mit komplexen Datentypen und unterstützen keine WSGUI-Konzepte über die Inferenz hinaus.

Die brauchbaren Ansätze zur GUI-Generierung aus Webservices sind der Übersichtlichkeit halber noch einmal in Tabelle 3.1 dargestellt.

Eine Verwendung der Implementierungen zu Testzwecken als Erweiterung des WSGUI-Editors dürfte aufgrund der mangelnden Konzeptunterstützung nicht ohne Modifikation der Software funktionieren. Somit wird auch Quelloffenheit zu einem Selektionskriterium. Der an sich recht professionelle Auftritt des Forms Generator und des SOAPscope hätte sicherlich zu guten Kandidaten geführt, die aber mangels Erfüllung des Kriteriums aus dem Rennen genommen werden mussten.

3.6. Analyse von Webservice-Beschreibungen

Der zu entwickelnde Editor hätte direkt basierend auf den vorhandenen Spezifikationen, insbesondere GUIDD, entwickelt werden können. Im Rahmen dieser Arbeit empfahl es sich aber zur Unterstützung des Autorenprozesses, einmal nachzuschauen, ob und inwiefern bereits Möglichkeiten der Annotation von WSDL-Dateien und den darin enthaltenen Schema-Daten genutzt werden, um diese wiederverwenden zu können.

Aus diesem Grund wurden möglichst viele WSDL-Dateien gesucht, die ein repräsentatives Abbild von realen Diensten darstellen. Das Auffinden von Diensten anhand gegebener Kriterien ist unter dem Begriff *Entdeckung* (*discovery*) ein Schwerpunkt der Forschung zu Webservices und hat dabei viele verschiedene Methoden hervorgebracht. Oftmals wird dabei *Universal Description, Discovery and Integration* (UDDI) [32] aufgeführt, da es selbst über Webservices implementiert ist. Als Verzeichnisdienst für die Suche wurde allerdings nicht etwa ein UDDI-Server ausgewählt, sondern der Anbieter *XMethods* [80], welcher neben UDDI auch weitere Formate zur Auflistung der registrierten Dienste unterstützt. Unter diesen befindet sich auch das Format Discovery Protocol, kurz DISCO, welches mit wenig Aufwand nutzbar ist und es ermöglicht, automatisiert sämtliche enthaltenen WSDL-Dateien aufzuspüren und herunterzuladen [44]. DISCO ist dabei ein einfaches Dateiformat, welches sich für jeden Host im Hauptverzeichnis der Webpräsenz befindet und somit gegenüber UDDI eine eher dezentrale Methode der Lokalisierung von Diensten propagiert. Zwar gilt DISCO mittlerweile als durch WS-Inspection abgelöst, für

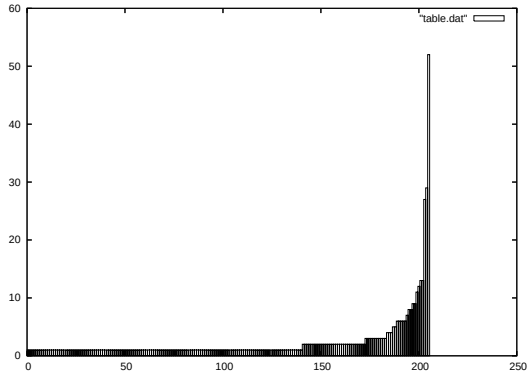

Abbildung 3.6.: Anzahl der WSDL-Dateien pro Dienstanbieter

die Lösung der vorliegenden Aufgabe war seine Einfachheit allerdings vorteilhaft.

Zum Zeitpunkt des Experiments waren 488 Dienste mit einer WSDL-Datei registriert, die von insgesamt 206 verschiedenen Anbietern stammten. Davon konnten 53 nicht heruntergeladen werden. Die große Mehrzahl der fehlerhaften Dateien, nämlich 46, wurde allerdings nicht mit einem HTTP-Fehlercode 404 quittiert, so dass diese Dateien unbeanstandet heruntergeladen wurden, obwohl sich hinter dem Namen nun Fehlerseiten im HTML-Format sowie Domainumleitungen zu kommerziellen Angeboten befanden. Eine derart hohe Fehlerrate dürfte für den Aufbau einer dienstorientierten Architektur (SOA) nicht förderlich sein, zumal zwei weitere Validierungsrunden die Zahl der nutzbaren Dienst weiter reduzieren dürfte: es muss das enthaltene Schema auf Validität geprüft werden, und schließlich die Funktionalität des Dienstes, da viele WSDL-Dateien Aufruf-URLs enthalten, die längst nicht mehr gültig sind. Details einer ähnlichen Analyse mit gleichwohl ernüchternden Zahlen finden sich in [7].

Die vorliegende Analyse verlangte hingegen nur die Wohlgeformtheit der XML-Struktur, die bei allen WSDL-Dateien gegeben war. Zurückzuführen ist das teilweise sicherlich darauf, dass ein großer Teil der Dateien automatisch generiert wurde. Hinweise auf entsprechende Toolkits wie Apache Axis oder Microsoft SOAP Toolkit finden sich implizit über die angegebenen (aber nicht verwendeten) Namespaces.

Die somit verbleibenden 435 Dienste sind in der Abbildung 3.6 bezüglich ihrer Anbieter dargestellt, wobei die starke Diskrepanz zwischen wenigen Anbietern mit vielen Diensten und vielen Anbietern mit nur ein oder zwei Diensten auffällig ist. Eine erneute Analyse oder die Einbeziehung anderer Anbieter könnten diese Verteilung sicherlich glätten, für die Erfüllung der Aufgabe hingegen soll die statistische Relevanz als gegeben angesehen werden.

Das Ergebnis ist recht erfreulich: 227 Dienste enthalten eine Beschreibung (52%), und von den insgesamt 4505 Operationen sind sogar 3569 dokumentiert (79%). Allerdings sind nicht wenige von den Texten im HTML-Format gehalten, was zwar gemäß der WSDL-Dokumentation zulässig ist, aber sicherlich nicht im Sinne einer Weiterverarbeitung als sinnvoll betrachtet werden kann. Eine Metainformation über die Art der Dokumentation

wäre hier wünschenswert.

Quasi überhaupt nicht dokumentiert sind hingegen die verwendeten Schemata. Nur zwei WSDL-Dateien enthalten menschenlesbare Kommentare zum Verwendungszweck der einzelnen Nachrichtenparameter, eine dritte enthält Annotationen für eine bestimmte Anwendung.

Dies verdeutlicht die Problematik, dass Dienste nicht ohne Zusatzinformationen oder Kenntnis der internen Nachrichtenstruktur dynamisch aufgerufen werden können, und betont den Einsatz eines WSGUI-Editors, welcher vorhandene Dokumentation einbinden kann, andererseits aber dem Nutzer freie Hand bei der Annotation aller XSD-Nachrichtenelemente und WSDL-Metainformationen lassen sollte.

3.7. Vorarbeiten zur Datenmodellierung

Neben dem Vorhandensein von Kommentaren in WSDL-Dateien sind auch Informationen über den Nachrichtenaufbau in Hinblick auf den Editor wichtig. Es werden nun zuerst Sprachen zur Beschreibung von Nachrichten, also von XML-Instanzen, vorgestellt werden. Die Generierung einer GUI aus solchen Modellen mit dem Zweck, die Instanz von der GUI erzeugen zu lassen, ist Teil der Betrachtungen.

3.7.1. Schema-Beschreibungssprachen

Die Aufgabe eines Schemas ist es, XML-Dokumente strukturell und inhaltlich zu beschreiben, so dass die Dokumente gegen das Schema validiert werden können. Man spricht in dem Zusammenhang auch von Instanzdokumenten, während das Schema jeweils eine Klasse von Dokumenten repräsentiert, nämlich die Menge aller validierenden Instanzen.

Schemasprachen im Bereich Webservices spezifizieren die XML-formatierten Nachrichten, die in der Ein- und Ausgabe eines Dienstes auftreten, beispielsweise verpackt als SOAP-Nutzdaten. In WSDL-Dateien wurde grundsätzlich die Möglichkeit geschaffen, beliebige Sprachen zur Beschreibung der XML-Daten zu nutzen. In der Praxis ist jedoch außer XML Schema (XSD) keine andere Schemasprache anzutreffen.

Die möglichen Gründe dafür können fehlende Werkzeugunterstützung für die Alternativen genauso einschließen wie eine objektive Überlegenheit von XSD. Ohne eine Wertung geben zu wollen, sollen an dieser Stelle ein paar der Alternativen vorgestellt werden.

DTD

Die *Document Type Definition (DTD)* wird allgemein als das Vorgängerformat von XML Schema angesehen. Es stammt aus dem SGML-Bereich, also dem Vorläufer von HTML-Dateien im Bereich Dokumentenerstellung, und wird in einer textuellen, nicht XML-basierten Syntax ausgedrückt. Der Aufbau und die Verwendung einer DTD ist innerhalb der XML-Spezifikation [13] beschrieben, was zur hohen Verbreitung beigetragen hat.

Das DTD-Format ist bis auf eher theoretische Konstrukte eine echte Teilmenge von XML Schema und soll somit nicht weiter betrachtet werden. Insbesondere fehlen jegliche Ansätze zur Typisierung von Daten, so dass eine GUI-Generierung ohne Mehraufwand

gegenüber XML Schema nicht durchführbar wäre. Auch Namespace-Unterstützung ist in DTDs aufgrund ihres Alters nicht gegeben.

TREX

Einen eher an reguläre Ausdrücke erinnernden Ansatz verfolgt das Format *Tree Regular Expressions for XML* [18], kurz *TREX*. Bei der Entwicklung war eine leichte Lesbarkeit ein wichtiges Kriterium. Der folgende Vergleich einer Elementdeklaration verdeutlicht dies.

```
<!-- XSD: Numerische Angaben der Häufigkeit -->
<xsd:element name="mindestens-eins" minOccurs="1"
   maxOccurs="unbounded" />
<!-- TREX: Symbolische Angabe -->
<trex:OneOrMore><trex:element
   name="mindestens-eins"/></trex:OneOrMore>
```

Da TREX bereits als überholt angesehen ist, soll nicht weiter darauf eingegangen werden. Es gibt aber sowohl etliche Beispieldateien als auch funktionierende Implementierungen dafür, so dass es sicherlich als einfache und konservative Alternative zu XSD dienen kann.

RELAX NG

Die Schemasprache RELAX NG (*RELAX Next Generation*) ist der gemeinsame Nachfolger von RELAX (*Regular Language Description for XML*) und TREX. Um die Wahl zwischen einer XML-basierten Syntax und einer textuellen Syntax nicht zu erschweren, liegt RELAX NG in einer abstrakten Syntax vor, von der Projektionen auf XML und Textformate existieren [39]. Die Verwandtschaft mit XML Schema auf der einen und mit DTD auf der anderen Seite ist dadurch unübersehbar.

Da RELAX NG nach XML Schema veröffentlicht wurde und dessen Typsystem verwenden kann, sind die Unterschiede zumeist struktureller Natur. Dabei sind sowohl mächtigere als auch schwächere Analogien zu XSD vorhanden. So ist die Angabe der Kardinalität von Elementen nur grob durchführbar, während Abhängigkeiten im Auftreten von Attributen eher feingranular ausdrückbar sind. Der folgende Ausschnitt verdeutlicht dies:

```
<!-- XML Schema: Beschränkung der Elemente auf fünf bis sieben
     Wäre in RELAX NG nicht ausdrückbar -->
<xsd:complexType>
  <xsd:element name="five-to-seven" type="xsd:int" minOccurs="5"
     maxOccurs="7"/>
</xsd:complexType>

<!-- Grenzen des Ansatzes von RELAX NG: eins bis unendlich viele
   Elemente -->
<rng:OneOrMore>
  <rng:element name="one-to-infinity"><rng:text/></rng:element>
</rng:OneOrMore>
```

Deutlich hebt sich die generischere Syntax von XSD von der eher sprachorientierten RELAX-NG-Syntax ab. Weniger offensichtlich, aber dennoch bemerkenswert, ist hingegen die Möglichkeit der Gruppierung von Attributen und Elementen, was in XSD nicht möglich ist:

```
<!-- RELAX NG: Beschränkung der Attribute
     Wäre in XML Schema nicht ausdrückbar -->
<rng:element>
  <rng:choice>
    <rng:group>
      <rng:element name="one".../>
      <rng:attribute name="two".../>
    </rng:group>
    <rng:attribute name="one-and-two".../>
  </rng:choice>
</rng:element>

<!-- Grenzen des Ansatzes von XML Schema: Nur Elemente in der
   Auswahl -->
<xsd:complexType>
  <xsd:choice>
    <xsd:group>
      <xsd:element name="one".../>
      <xsd:element name="two".../>
    </xsd:group>
    <xsd:element name="one-and-two".../>
  </xsd:choice>
</xsd:complexType>
```

Letztendlich kann man durch eine Kombination der Schwächen speziell präparierte XML-Dokumente erstellen, die zwar einfach und intuitiv zu verstehen sind, aber sowohl durch XML Schema als auch durch RELAX NG nicht ausdrückbar sind. Aus objektiver Sicht sind also beide Formate nicht optimal in ihrer Mächtigkeit.

Weitere Ansätze

Neuere Schemasprachen wie *Document Structure Description* sind mittlerweile spezifiziert, um Unzulänglichkeiten in den existierenden zu beheben. Parallel dazu gibt es Erweiterungen zu XML Schema wie etwa *SchemaPath* [52], um die im Beispiel gezeigten Grenzen des gegenseitigen Ausschlusses von Elementen und Attributen, sogenannten *co-occurrence constraints*, zu überwinden. Zusätzlich zu den aufgeführten grammatikbasierten Sprachen gibt es auch noch regelbasierte, von denen *Schematron* wiederum in XML Schema integriert werden kann, ohne dass existierende XSD-Parser dies jedoch verarbeiten können. All diese Ansätze sind noch relativ jung und werden somit einige Zeit brauchen, um eine gewisse Mindestrelevanz zu erreichen. Ihr Aufkommen zeigt aber, dass die Beschreibung von XML-Dokumenten nicht trivial ist und die Entwicklung von Sprachen dafür stetig voranschreitet.

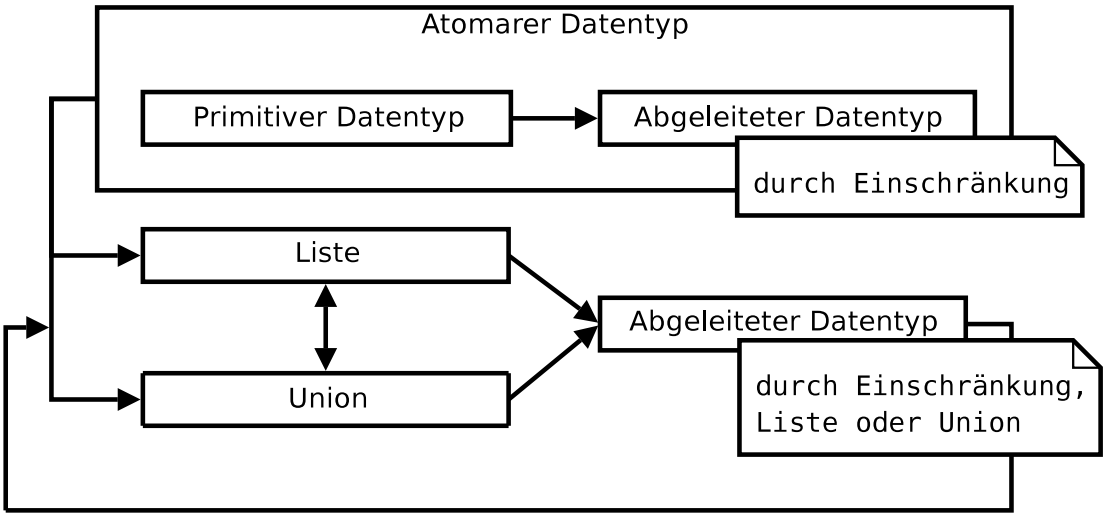

Abbildung 3.7.: Definition simpler Typen in XML Schema

3.7.2. Untersuchung von XML-Schema

Die Durchführung einer Abbildung abstrakter Datenmodelle auf GUI-Elemente bedarf stets einer Untersuchung des Datenmodells auf Eignung für diese Abbildung. Auf dem Gebiet der Webservices ist XML Schema dominant, und dient damit als Gegenstand der Untersuchung.

Der XSD-Standard beschreibt eine Typhierarchie, ausgehend von einem imaginären Ur-Typ, auf Basis nur weniger davon abstammenden primitiver Typen: Zeichenketten, Zahlen, Datumsformate, und spezielle XML-Typen. Alle weiteren Typen sind entweder Einschränkungen davon oder aber Zusammensetzungen der primitiven, möglicherweise eingeschränkten, Typen. Die primitiven Typen zusammen mit ihren eingeschränkten Abkömmlingen werden als einfache Typen bezeichnet (Abbildung 3.7), die zusammengesetzten als komplexe Typen (Abbildung 3.8).

XML Schema enthält etwa 45 eingebaute einfache Datentypen, die größtenteils voneinander abgeleitet sind. Es sind keine komplexen Datentypen enthalten, diese entstehen erst durch die Definition in einer XSD-Datei.

Einschränkungen auf einfachen Datentypen werden auch als Facetten bezeichnet. Diese teilen sich hauptsächlich in numerische Facetten wie minimaler und maximaler Wert oder Anzahl der Nachkommastellen sowie in Zeichenketten-Facetten wie maximale Länge an Zeichen oder Konformität mit einem regulären Ausdruck ein. Diese *regular expressions (regexps)* beschränken den Wertebereich einer Zeichenkette anhand eines erdachten finiten Automaten, dessen mögliche Produktionen in einer speziellen Syntax ausgedrückt werden, welche auch von vielen Programmiersprachen bekannt ist. Es tritt jedoch in XML Schema ein spezieller „Dialekt" auf, so dass man von einem Format im Format sprechen kann, und man für die Verarbeitung der entsprechenden Facetten am besten auf einen zusätzlichen Parser speziell für die regulären Ausdrücke zurückgreift.

Neben dem Typsystem enthält XML Schema noch einen strukturellen Aspekt, der die

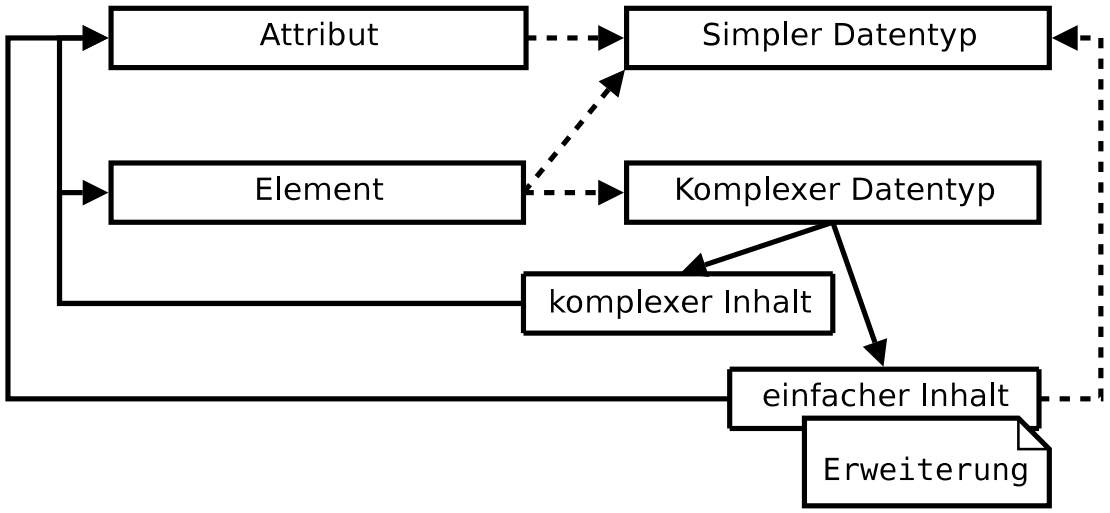

Abbildung 3.8.: Definition komplexer Typen in XML Schema

Beziehungen von Elementen, Attributen und Typen untereinander reglementiert. So ist es möglich, über eine Sequenz von XML-Elementen ein Attribut als Primärschlüssel festzulegen, so dass jeder Attributwert nur maximal einmal in allen Elementen zusammen vorkommen darf. Dieser aus der Datenbankwelt stammende Teil des Schema-Standards verwendet *XPath*-Ausdrücke [19] zur Adressierung der Elemente und Attribute. Die Einbindung von XPath führt dazu, dass nun also noch eine weitere Spezifikation bekannt sein muss, um XML Schema verarbeiten zu können.

Zusammengefasst ließen sich die für XML Schema relevanten Spezifikationen so darstellen wie in Abbildung 3.9 gezeigt.

3.7.3. Kompatibilität mit XForms

Die Formularbeschreibungssprache XForms nutzt XML Schema zur Repräsentation des Datenmodells. Es sind allerdings einige Einschränkungen und Erweiterungen am Umfang des Typmodells vorgenommen worden, so dass eine 1:1-Kompatibilität leider nicht sichergestellt ist.

Nicht unterstützt werden die Datentypen `xsd:duration`, `xsd:ENTITY`, `xsd:ENTITIES` und `xsd:NOTATION`. Als Ersatz für `xsd:duration` sind `xforms:dayTimeDuration` und `xforms:yearMonthDuration` hinzugekommen. Zusätzlich wurde noch der Typ `xforms:listItem` eingeführt, der in einem XML-Schema-Listenkontext als mehrfach auftretendes Element innerhalb eines `xforms:listItems` genutzt werden kann.

Zur Bewertung ist zu sagen, dass die Änderungen an `xsd:duration` nicht verständlich sind, da der Typ an sich bereits Zeitabschnitte im Subsekundenbereich genauso repräsentieren kann wie einen Zeitraum von mehreren Jahren. Die Einführung von `xforms:listItem` ist hingegen als Tugend in der Not als sinnvoll zu erachten, da XSD-Listen dadurch gebildet werden, dass man einfach Zeichenketten durch Leerzeichen getrennt hintereinander schreibt. Nun können uneingeschränkte Zeichenketten jedoch selbst Leer-

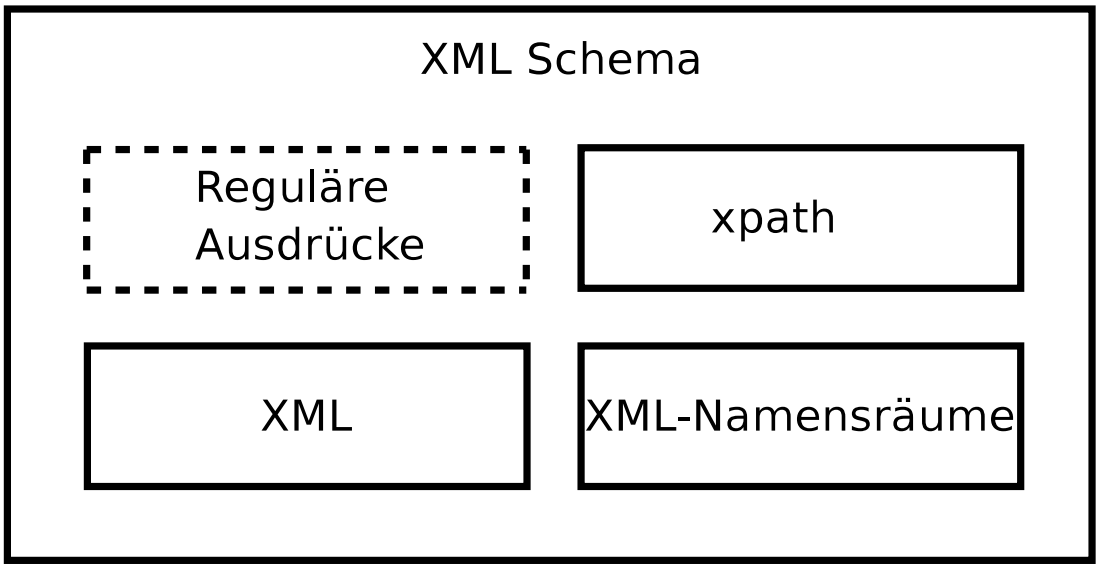

Abbildung 3.9.: Relevante Spezifikationen zur Verarbeitung von XML Schema

zeichen enthalten, und `xforms:listItem` verbietet das nun. Optimal hingegen wäre es gewesen, wenn der XSD-Standard von vornherein die Verwendung von `xsd:string` als Listentyp ausgeschlossen hätte.

Durch all diese beschriebenen Inkompatibilitäten müssen WSGUI-Prozessoren einen zwar nicht allzu großen, aber dennoch spürbaren Konvertierungsaufwand leisten. Um Fehler auszuschließen, sind deterministische Regeln notwendig.

3.7.4. Semantische Hinweise und Inferenz

Es wurde gezeigt, dass es oft nicht ideal ist, nur das Schema zur GUI-Generierung heranzuziehen. Die Untersuchung von XML Schema lieferte zwar das Ergebnis, dass sich Formularkomponenten für nahezu jeden Schematyp generieren lassen, aber es bleiben dennoch über die Typsicherheit hinaus gewisse Wünsche offen. Für ein semantisches Web bräuchte man Semantikhinweise zusätzlich zum Typsystem in XSD, die es aber nicht gibt [45].

So geht aus einem Schema nicht hervor, ob ein Feld mit dem Datentyp `string` und dem Variablennamen `id` nun eine Kennung beliebiger Natur oder aber eine vertrauliche Kennung, etwa ein Passwort, darstellen soll. Im zweiten Fall sollte ein GUI-Element zur Eingabe verwendet werden, welches die eingegebenen Zeichen nicht anzeigt, sondern maskiert. Derartige Passwort-Eingabefelder dienen nicht nur dem Sichtschutz, sondern auch darüberhinaus dem Datenschutz, da der Webbrowser ihren Inhalt nicht über den Seitenaufruf hinaus speichert, während er das bei herkömmlichen Eingabefeldern zum Zweck der Effizienz tun kann.

In einer GUIDD-Datei könnte nun ein entsprechendes Passwort-Eingabefeld als GUI-Element für das betroffene Datenfeld vermerkt werden. Während Formularkomponenten

explizit einer WSGUI-Engine vorschreiben, welches GUI-Element sie für das jeweilige Nachrichtenelement zu verwenden hat, gäbe es aber auch die Möglichkeit, nur Hinweise auf Beschriftung und Semantik des Datenfeldes mitzugeben.

Der Vorteil wäre die universelle Anwendbarkeit der Hinweise unabhängig von der Klasse der verwendeten GUI-Elemente in den Formularkomponenten. Der Nachteil wäre ein höherer Arbeitsaufwand innerhalb der WSGUI-Engine zur Zusammensetzung der GUI-Elemente sowie die Pflege eines weiteren Datenformats. GUIDD-Dateien enthalten momentan keine Möglichkeit, semantische Informationen einzutragen. Durch den Editor sollte es jedoch möglich sein, durch explizite Angabe von Formularkomponenten beispielsweise die Eingabe in einem Passwortfeld zu erzwingen.

4. Konzeption des Editors

Vor dem Beginn der eigentlichen Entwicklungsaufgaben am geplanten Editor stand eine umfangreiche Recherche zu den notwendigen Eigenschaften und zu Möglichkeiten zu deren Umsetzungen auf den verfügbaren Laufzeitplattformen unter Verwendung der bereits besprochenen Standardformate zur Oberflächenbeschreibung. Neben WSGUI-Konzepten sollten dabei auch Editorkonzepte beachtet werden, so dass sich die Anforderungen an den Editor durch eine eine Verzahnung zweier Anforderungslisten herleiten ließ.

4.1. Anforderungen

Ein grafischer Editor sollte vom Benutzer einfach und intuitiv bedient werden können, gleichzeitig aber mächtig genug sein, möglichst viele Einsatzszenarien abzudecken. Um eine Mehrarbeit zu vermeiden, war es aus diesem Grund notwendig, neben einer nativen Integration der Oberflächengenerierung für eine Plattform weitere Szenarien durch einen Dateiexport und die Möglichkeit der Vorschau durch zusätzliche Software zu ermöglichen. Das Merkmal des Dateiaustauschs gilt insbesondere für Übersetzungen, da diese Tätigkeit als von der des Editierens von WSGUI-Hinweisen getrennt angesehen werden kann.

Der Editor selbst sollte plattformübergreifend arbeiten, wobei eine modulare Struktur helfen soll, zusätzliche Funktionalität je nach Plattform hinzuzufügen, anstatt allen Plattformen die Schnittmenge an Funktionalität zur Verfügung zu stellen. Für die Implementierung wird also ein plattformübergreifendes Framework gesucht, welches möglichst bereits Basisfunktionalität zur Verarbeitung von den relevanten XML-Dialekten XML Schema und XForms bietet.

Die verschachtelte Struktur komplexer Webservice-Nachrichten (*WSDL messages*) legt es nahe, eine baumartige Ansicht für die jeweils zu editierende Nachricht zu gestalten. Dies entspricht nicht unbedingt dem späteren direkten Aufruf des Dienstes, der beispielsweise durch ein zu kleines Display in mehrere Abschnitte unterteilt werden könnte, oder generell für jede Teilnachricht einen neuen Dialog öffnen würde. Eine Vorschrift zur Umwandlung eines Schemas in eine solche Baumansicht, also in eine spezielle kanonische Instanz des Schemas, muss dazu erarbeitet werden.

Zwar sind XML-Instanzen per se nicht auf Baumstrukturen beschränkt: [58]

> Allen diesen Untersuchungen liegt nach wie vor die Annahme zugrunde, dass jedes annotierte Dokument seiner Struktur nach ein Baum ist. Diese Annahme ist jedoch nur in erster Näherung korrekt. Zwar definiert der Grammatikformalismus für eine DTD, daß jedes Dokument ein Element ist, und daß Elemente nur nebeneinander stehen können oder ein Element ein anderes vollständig beinhaltet, so daß Überlappungen ausge- schlossen sind, und als

> Grundstruktur ein Baum entsteht. Aber Attribute vom Typ ID und IDREF erlauben durch Verweise den Aufbau einer weiteren Struktur, die klar über Bäume hinaus zu allgemeinen Graphen geht. Damit werden Komplexitätsresultate der Graphentheorie relevant.

Diese „erste Näherung" soll für den WSGUI-Editor jedoch ausreichend sein, so dass eine Baumansicht der Instanz gerechtfertigt ist [25].

Schließlich sind auch noch die Typrestriktionen der Eingabefelder zu beachten, die von einfachen Datentypen im Schema abgeleitet wurden. Spezielle Softwarekomponenten helfen bei der Umsetzung dieser Anforderung. Typsicherheit in den erzeugten Dialogen sollte nicht vom Anwender des Editors unabsichtlich übergangen werden dürfen.

Als generelle Anforderung an die Funktionalität steht ebenfalls der Wunsch, möglichst unabhängig von den konkreten WSGUI-Dateiformaten arbeiten zu können, und neue WSGUI-Konzepte, welche unter Umständen weitere Formate mit sich bringen, ebenfalls abdecken zu können. Die Mindestanforderung liegt dabei bei der Unterstützung des GUIDD-Formats (*Graphical User Interface Deployment Descriptor*), welches die Überlagerung einzelner Nachrichtenelemente in der Anzeige mit vorgegebenen Eingabefeldern und Bezeichnungen vorsieht.

Zusammenfassend liegen für die Konzepterstellung die nachfolgenden Anforderungen vor.

- Keine Voraussetzung von Spezialkenntnissen des Anwenders im XML-Umfeld
- Plattformübergreifende Funktionalität
- Vorschaufunktionalität und Datenexport
- Baumansicht einer kanonischen Instanz
- Gewährleistung der Typsicherheit
- Unterstützung des GUIDD-Formats, aber auch anderer WSGUI-Konzepte

4.2. Evaluation der Basisarchitektur

Die Entwicklung eines grafischen Editors ist eine immer wieder erforderliche Tätigkeit. Mehrere Arbeiten haben sich konkret mit Editoren beschäftigt, welche auf Basis von XML arbeiten. Ein paar davon werden am Ende des Kapitels vorgestellt werden. Trotz der Unterschiede im Anwendungsgebiet bleibt als Gemeinsamkeit festzustellen, dass jeder dieser Editoren die XML-Grundlage verbirgt und stattdessen eine konzeptorientierte Benutzerführung vornimmt, in deren GUI die XML-Spezifika versteckt wird.

Auch der WSGUI-Editor sollte diesem Vorbild folgen. Aus diesem Grund wird eine Modifikation bereits vorhandener XML-Schema-Editoren weniger sinnvoll sein als eine Neuentwicklung eines speziellen Editors. Ein weiterer Grund ist die Kombination der Verarbeitung von WSDL und XSD, welches durch die wenigsten Editoren geboten wird, von generischen XML-Editoren abgesehen. Für die Erstellung von WSGUI-Informationen

für einen Dienst sollte diese Unterscheidung von Dienst- und Operationsmerkmalen auf der einen und Nachrichtenmerkmalen auf der anderen Seite nicht sichtbar sein.

Gleichzeitig darf der Entwicklungsaufwand für den Editor nicht übermäßig ansteigen. Eine möglichst weitgehende Nutzung existierender Bibliotheken und Werkzeuge, die in den Editor eingebunden werden können, ist nunmehr erforderlich. Eine hohe Gewichtung haben hierbei Bibliotheken, die bereits für den Umgang mit Webservices ausgelegt sind. Man spricht hierbei auch von *Webservice-Stacks* oder *-toolkits*. Das Vorhandensein eines solchen Webservice-Stacks ermöglicht das Einlesen der WSDL-Dateien und die Entwicklung einer Testanwendung auf SOAP-Basis. Für die Bearbeitung der Nachrichtenmerkmale ist ein XSD-Parser darüber hinaus unverzichtbar.

Nach einer Recherche konnten vier Entwicklungsplattformen identifiziert werden, welche mit einem ausreichend entwickelten Webservice-Stack ausgestattet sind. Es sind dies Eclipse, Apache, KDE/Qt und Mozilla.

Eclipse ist eine integrierte Entwicklungsumgebung (IDE), die über Plug-ins alle möglichen Zwecke erfüllen kann. Sie enthält einen Editor, einen Java-Compiler und verschiedene Werkzeuge für die Arbeit mit XML. Der Editor ist sehr mächtig und kann aspektabhängig im Erscheinungsbild geändert werden. Für die Darstellung nutzt Eclipse SWT, eine Java-Anbindung an das jeweilig dominante GUI-Toolkit für jede Zielplattform [22]. Zusätzliche Funktionalität lässt sich über Plugins zu Eclipse hinzufügen.

Apache ist ein Projekt, welches mit dem gleichnamigen HTTP-Server bekannt geworden ist. Das Projekt pflegt jedoch auch eine hohe Anzahl von Komponenten im Bereich Middleware und Webservices, welche hauptsächlich in Java geschrieben sind. Oft wurden diese Komponenten beigesteuert, dennoch ist der Grad an Integration recht hoch. Axis ist eine solche Komponente, die als Servlet im Servlet-Container Tomcat läuft und zur Implementierung von Servern, aber auch Clients, für Webservices eingesetzt wird [56][21]. Als Serverprojekt würde Apache als GUI-Äquivalent Webanwendungen ermöglichen.

KDE ist eine auf der Klassenbibliothek Qt aufbauende Arbeitsoberfläche, die neben Büroprogrammen, Webbrowser und sonstigen Anwendungen auch Werkzeuge für Entwickler enthält [23]. So ist das Kode-Projekt enthalten, welches einen Webservice-Stack sowie den bereits vorgestellten Aufrufer Kung enthält.

Das Mozilla-Projekt [64] ist hauptsächlich bekannt für die Webbrowser Mozilla und Firefox sowie verwandter Anwendungen wie E-Mail-Client, HTML-Editor oder Kalender. Die technische Umsetzung der GUI geschieht über XUL bzw. XPFE. Neue Anwendungen können hinzugefügt werden, indem eine Menge an XUL-Dateien mitsamt Anwendungslogik in JavaScript als Paket nachinstalliert werden. Das Vorhandensein eines HTML-Editors lässt die Idee aufkommen, die Mozilla-Plattform als Grundlage für den WSGUI-Editor zu nutzen. Speziell für die Unterstützung der Anwendungsentwicklung wurde die XPFE-Laufzeitumgebung aus dem Browser herausgetrennt und kann separat als *XULrunner* eingesetzt werden.

Die Tabelle 4.1 fasst die untersuchten Plattformen und ihre Webservice-Stacks zusammen.

Die Verwendung von Apache Axis ist durch die Fokussierung auf Serveranwendungen zumindest im Rahmen dieser Arbeit als ungeeignet für den WSGUI-Editor zu betrachten. Zwar wäre es denkbar, den Editor als Webanwendung zu entwickeln, eine Handhabung

Plattform	WS-Stack	Sprache	Umgebung	Editorfähigkeit
Eclipse	(verschiedene)	Java	Desktop/IDE	ja
Apache	Axis	Java	Server	nein
KDE/Qt	Kode	C++ und andere	Desktop	ja
Mozilla	(verschiedene)	C++	Desktop/Browser	ja (begrenzt)

Tabelle 4.1.: Plattformen für Entwicklung von Webservice-Werkzeugen

wäre jedoch wesentlich unkomfortabler als mit einer Desktopanwendung. Durch den Einsatz von Techniken, die häufig unter dem Namen AJAX (*Asynchronous JavaScript And XML*) zusammengefasst werden, kann man den Komfort zwar steigern, doch dies wäre in der gegebenen Zeit kaum zufriedenstellend implementierbar.

Eine Implementierung als Mozilla-Programm unter Nutzung der Oberflächensprache XUL wäre theoretisch möglich, die stark verwobenen Interna der von den Mozilla-Browsern verwendeten Bibliotheken für WSDL und XSD machte dies allerdings schwierig. Die fehlende Dokumentation der betroffenen Bibliotheken und der doch relativ hohe Ressourcenbedarf von Mozilla-Anwendungen sprechen ebenfalls gegen einen Einsatz dieser Plattform.

Sowohl die Verwendung von Eclipse als auch die von KDE/Qt sind beide grundsätzlich interessant. Eclipse bietet mit dem EMF (*Eclipse Modeling Framework*) bereits eine Komponente für die Editorentwicklung an. Andererseits steht auch hierbei, ähnlich wie bei Mozilla, die Frage nach der Zweckmäßigkeit der Verwendung einer großen Plattform im Raum. Eine Arbeit zu dem Thema [70] beschreibt die Nutzung des EMF für die Erstellung einer Anwendung im Bereich modellgetriebene Entwicklung. Für den WSGUI-Editor ist jedoch auch die Verfügbarkeit von Webservice-Technologie entscheidend.

Sowohl Eclipse als auch KDE bieten einen Aufrufer für Webservices, der in beiden Fällen nur mit einfachen Diensten arbeiten kann und keinerlei WSGUI-Hinweise unterstützt. Eine Überarbeitung des Aufrufers als externe Anwendung oder integrierte Komponente für die Vorschau wäre also auf alle Fälle notwendig. Da Eclipse sich hauptsächlich an Entwickler richtet, der Editor aber als alleinstehende Anwendung konzipiert wurde, die auch von weniger technisch versierten Menschen bedient werden soll, wird von der Nutzung der Eclipse-Plattform zur Entwicklung des WSGUI-Editors trotz deren guter Integration von Editorframework und Webservice-Standards abgesehen. Die Existenz eines WYSIWYG-Editors auf der KDE/Qt-Plattform und dessen Erweiterbarkeit mit eigenen GUI-Elementen gab letztendlich den Ausschlag in Richtung KDE/Qt als Basis für die Implementierung.

Diese Wahl impliziert eine Erweiterung des Aufrufers Kung um die Fähigkeit, GUIDD-Dateien einlesen zu können, um ihn als Vorschaukomponente nutzen zu können. Somit können sämtliche Bestandteile des Editors wie aus einem Guss verwendet werden, auch wenn die Herkunft all dieser Qt-basierten Anwendungen höchst unterschiedlich ist.

4.3. Architekturbeschreibung

Die getroffene Wahl der Basisarchitektur soll nach der kurzen Vorstellung im Rahmen der Evaluation nun genauer vorgestellt werden. Dabei werden die verfügbaren Komponenten und deren Nutzbarkeit für den Editor besprochen.

Als GUI-Toolkit kommt auf der KDE/Qt-Plattform die Bibliothek *Qt* zum Einsatz, die im Abschnitt über deklarative GUI-Beschreibungen bereits mit ihrem eigenen Format Qt UI vorgestellt wurde.

Qt als Bibliothek ermöglicht es Anwendungen, ohne Quelltextmodifikationen auf Unix-ähnlichen Betriebssystemen wie Linux oder Solaris, auf Windows und auf Mac OS X zu laufen, wobei dennoch aus Gründen der Effizienz für jedes System eine individuelle Compilierung stattfinden muss. Darüber hinaus steht eine spezielle Edition unter dem Namen *Qtopia Core* für mobile Geräte unter Linux zur Verfügung. Diese wurde im Rahmen einer studentischen Teamarbeit an der TU Dresden auch auf das auf Sicherheit und Echtzeitanwendungen spezialisierte Betriebssystem L4 übertragen, so dass die Portabilität auch auf weitere Systeme kein Hindernis darstellen sollte.

In der aktuellen Version ist Qt in verschiedene Module gegliedert, die sich nahezu beliebig kombinieren lassen. Es existiert ein Kern (`QtCore`), eine Sammlung von GUI-Widgets und Dialogen (`QtGui`), Routinen zur XML-Verarbeitung (`QtXML`) und eine Implementierung von Internetprotokollen (`QtNetwork`). Darüber hinaus stehen noch Datenbankanbindungen, 3D-Grafikroutinen und ähnliches bereits. Diese für den WSGUI-Editor nicht benötigten Bestandteile lassen sich aber effektiv aus der Kompilierung herausnehmen.

Anwendungen, die unter Verwendung von Qt geschrieben sind, lassen sich vor allem im Umfeld netzwerktransparenter Systeme finden. *Skype* etwa ist ein Programm zur Internettelefonie, *NX Client* bietet grafischen Zugriff auf entfernte Rechner über das komprimierende NX-Protokoll, und das *K Desktop Environment* (KDE) nutzt Qt zur grafischen Darstellung, aber auch für die Implementierung virtueller Dateisysteme auf Basis von Internet-Protokollen. Das dafür verwendete KIO-Framework [28] erlaubt es, beliebige Datenübertragungswege zu implementieren, so dass diese für jede Anwendung zur Verfügung stehen und transparent in den Dateidialogen und sonstigen Dateiroutinen synchron und asynchron genutzt werden können. Für den WSGUI-Editor wäre eine Umsetzung eines Protokolls denkbar, welches es ermöglicht, GUIDD-Dateien neben WSDL-Dateien in einem Verzeichnisdienst zu veröffentlichen. Es würde dann kein spezielles Programm zum Publizieren mehr benötigt, und der Arbeitsablauf damit optimiert. Um beispielsweise UDDI-Verzeichnisse anzusprechen, müsste das Protokoll nur wenige Befehle auf SOAP-Basis implementieren.

Um die Effizienz zu wahren und bereits existierende Komponenten für dieses Toolkit verwenden zu können, fiel die Wahl der Programmiersprache für den WSGUI-Editor auf C++. Es sei an dieser Stelle aber mit vermerkt, dass Teilaufgaben auch mit schwach typisierten Skriptsprachen wie etwa Python durchführbar sind, und somit die Entwicklungszeit spürbar gesenkt werden kann, was gerade in der Phase der Prototypisierung und dem damit einher gehenden Verwerfen von Lösungen sinnvoll ist. Qt bietet neben C++ Sprachanbindungen für Python, Ruby und Java an, wobei nicht alle Bibliotheken mit diesen Anbindungen versehen sind. Die Nutzung von C++ verschafft die größtmögliche

Flexibilität bei der Wahl der Qt-basierten Komponenten.

4.3.1. Existierende Komponenten

Für die Suche nach Komponenten sind in erster Linie die auf Qt aufbauenden Anwendungen und Entwicklungsumgebungen relevant. Das Programm Kung wurde bereits vorgestellt. Andere Werkzeuge in dessen Umfeld und weitere nützliche Programme werden nun präsentiert.

Das auf dem Qt-Toolkit basierende *K Desktop Environment* (KDE) [23], eine freie Desktopumgebung, ist in verschiedene Teilprojekte untergliedert. Für die Entwicklung des Editors standen dabei vor allem zwei davon in der engeren Wahl einer Analyse der bereits vorhandenen Technologien, nämlich *kdenetwork* (Programme für den Einsatz im Netzwerk) und *kdepim* (Persönliche Informationsverwaltung und Diensteintegration), jeweils mit den dazugehörigen Bibliotheken. Innerhalb von *kdepim* befindet sich das *kode*-Framework [69], welches letztendlich aufgrund von Kung und den Webservice-Bibliotheken die tragende Rolle für die Entwicklung des Editors zugewiesen bekommen hat und demnach an dieser Stelle kurz vorgestellt werden soll.

Kode ist entstanden als Code-Generierungsprojekt, welches aus XML-Schemata entsprechende C++-Klassen generiert, die Zugriffe auf Instanzen entsprechend dieser Schemata auf für Programmierer komfortable Art und Weise ermöglichen. Ähnliche Ansätze finden sich in der Java-Welt etwa mit dem Programm `javaCC`. Neben dem Code-Generator mit dem Namen `kode_bin` enthält Kode aber mittlerweile weitaus mehr Bestandteile aus dem XML- und Webservices-Umfeld. Wichtig für den Editor sind eine Bibliothek für die Arbeit mit XML Schema sowie mit WSDL, zu finden in der Bibliothek KWSDL.

KSchema ist die XML-Schema-Implementierung von Kode. Es ist eine Bibliothek, die für das Nachladen von Schema-Dateien von Servern auf das KIO-Framework aufsetzt und somit transparent für den Programmierer miteinander verknüpfte Schemata einlesen kann.

KWSDL ist die Basisbibliothek für Webservices unter KDE. Sie bietet einen Parser für WSDL-Dateien sowie einen für die Konversations- und Kompositionssprache WSCL an, und desweiteren ist die Anwendung Kung hier zu finden.

Kung ist ein Desktop-Programm, das es erlaubt, Webservices dynamisch aufzurufen, ähnlich vergleichbaren Webanwendungen wie beispielsweise Dynvoker. Die Nachrichten der aufgerufenen WSDL-Operation werden dabei zum Editieren angezeigt und anschließend die Ergebnisnachricht dargestellt. Von der Anwendung selbst wurde dabei noch eine Bibliothek abgespalten, welche all jene Klassen enthält, die zum Editieren von Datenfeldern notwendig sind.

Zur Illustration sei an dieser Stelle noch einmal in Abbildung 4.1 die Hierarchie der Bibliotheken und Anwendungen dargestellt.

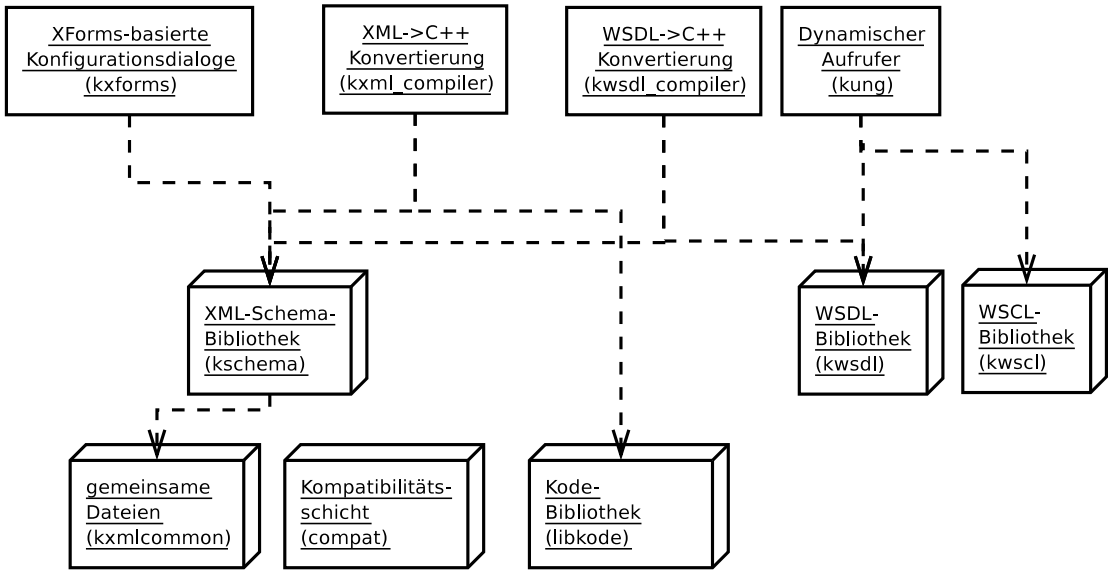

Abbildung 4.1.: Architektur des Kode-Projekts für XML- und WSDL-Anwendungen

Neben dem Kode-Framework bietet KDE auch nützliche Anwendungen an, wie beispielsweise den XML-Editor `kxmleditor`. Eine Integration solcher existierenden Programme in den WSGUI-Editor führt zu sogenannten schwachen Abhängigkeiten, das heißt, die Installation der Programme ist optional und eine Nutzung derselben kann je nach Installationsumfang ohne Rekonfiguration aus dem WSGUI-Editor heraus erfolgen.

Aus dem Funktionsumfang von Qt selbst stammen der bereits vorstellte WYSIWYG-Editor Qt Designer sowie Qt Linguist zur Bearbeitung von Übersetzungen. Beide Werkzeuge sind für sich genommen Editoren, deren Einbindung in den WSGUI-Editor zweckmäßig erscheint.

4.3.2. Verwendung der Komponenten

Das Kode-Framework enthält bis auf die Unterstützung von GUIDD-Dateien alle benötigten Grundlagen, um den WSGUI-Editor konstruieren und integrieren zu können. Die Abbildung 4.2 erläutert die Nutzung der Bestandteile für den Editor.

Im Gegensatz zu Kung werden die Klassen zur Komposition von Webservices nicht benötigt. Auch erfolgt die Darstellung nicht als verschachtelte Widgets, sondern in einer der XML-Struktur gerecht werdenden Baumdarstellung, in denen einzelne Bestandteile auf- und wieder zugeklappt werden können. Die übrigen Bestandteile von Kode wie etwa der XML-Compiler werden ebenfalls nicht benötigt. Ihr Einsatzgebiet ist die Quelltexterzeugung im Rahmen der modellgetriebenen Entwicklung und nicht die GUI-Generierung.

Damit Kung als Vorschauanwendung nutzbar wird, muss die Anwendung mit einer Unterstützung für GUIDD-Dateien ausgestattet werden. Dazu muss eine Bibliothek erstellt werden, die ähnlich dem Java-Pendant GUIDD4J [73] einen Parser für das GUIDD-Format sowie eine Zugriffsmöglichkeit auf die in Objektform gebrachten Informationen

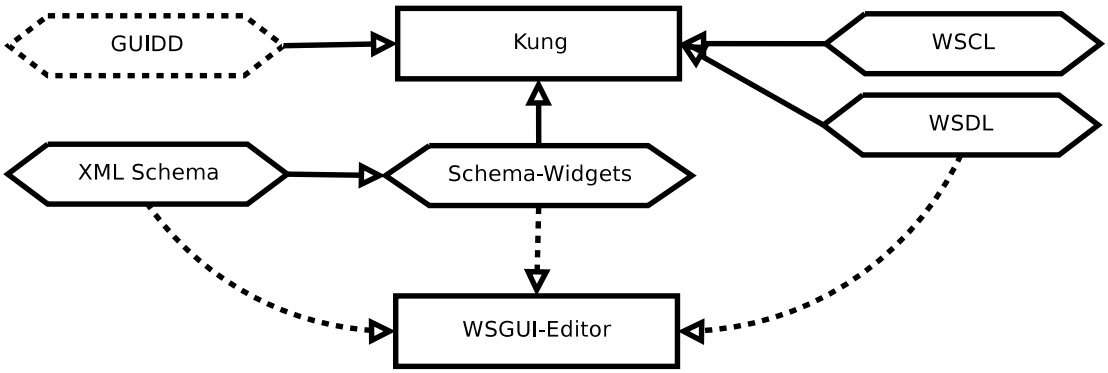

Abbildung 4.2.: Wiederverwendung der Bibliotheken für den Editor

aus der GUIDD-Datei enthält. Ein Vorschaudialog sollte desweiteren kein Absenden der Daten zulassen.

Sämtliche Abhängigkeiten auf KDE-Frameworks wie KIO sollten abgeschwächt werden, um den Installationsaufwand gering zu halten und (bis auf Kode) mit einer reinen Qt-Basis arbeiten zu können. Der Qt Designer benötigt als WYSIWYG-Editor für XForms-basierte Oberflächen entsprechende Widgets, die als Erweiterung installiert werden müssen. Existierenden Widgets fehlt vor allem die Kopplung des Eingabefeldes an die Beschriftung, wie von XForms vorgesehen.

Die Anpassungsarbeiten für die Verwendung der existierenden Komponenten vor der eigentlichen Entwicklung des Editors lassen sich also wie folgt zusammenfassen:

- Modifikation von KSchema zur Verwendung der `QtNetwork`-Klassen statt KIO
- Entwicklung einer GUIDD-Bibliothek für C++
- Vorschaumodus und Nutzung der GUIDD-Bibliothek in Kung
- XForms-Widgets als Erweiterung zu Qt Designer

4.4. Unterstützung für WSGUI-Konzepte

WSGUI ist eine formatübergreifende Bezeichnung für die Anreicherung von puren maschinenlesbaren Webservice-Schnittstellenbeschreibungen mit Angaben, die die Nutzung durch Menschen möglich machen. Dazu gehören Beschriftungen mitsamt deren Übersetzungen, Hinweise auf die semantische Verwendung von Werten über ihre reine syntaktische Typisierung hinaus und weitere Angaben.

Welche der WSGUI-Konzepte für den Editor eine Rolle spielen und ob neben denen, die im GUIDD-Format ausgedrückt werden können, noch weitere unterstützt werden müssen, sollen die folgenden Unterabschnitte beleuchten. Für die Umsetzung im Editor wurde dafür eigens eine statistische Untersuchung von XSD-Merkmalen in Webservices durchgeführt, um schwieriger zu implementierende Konzepte nicht zu viel Entwicklungszeit

kosten zu lassen, wenn sie dann in der Praxis ohnehin nicht zum Einsatz kommen. Diese Untersuchung geht über die WSDL-Analyse im WSGUI-Kapitel hinaus und beinhaltet sowohl Struktur- als auch Typmerkmale von XML Schema.

4.4.1. Konzeptevaluierung

Es werden nun die bereits bekannten WSGUI-Konzepte noch einmal im Detail vorgestellt und deren Einbindung in den Editor diskutiert. Eine umfangreiche Dokumentation der Konzepte und deren Abhängigkeiten untereinander findet sich in einer geplanten Veröffentlichung [76].

Inferenz bezeichnet die Ableitung einer GUI aus einem formalen Modell. Dabei werden sowohl Typinformationen im Schema als auch auffindbare Dokumentation verwendet. Inferenz führt zu typsicheren GUIs, welche je nach Umsetzung auch die volle Flexibilität zur Ausreizung eines Schemas bieten. Allerdings führt es nicht zu allgemein nutzbaren Oberflächen. Mehr Informationen zu diesem Thema, speziell zur Inferenz von XForms-basierten Formularen aus XSD-Angaben, finden sich in einer Ausarbeitung [72].

Der Editor sollte Inferenz nutzen, um ein initiales Modell für die WSGUI-Hinweise einzublenden. Editiert der Nutzer explizit die jeweiligen Einstellungen, haben die Vorgaben des Nutzers Präferenz vor den durch Inferenz erhaltenen Voreinstellungen.

Gleichzeitig dient die Typinferenz dazu, die Interaktion des Nutzers einzuschränken. So sollte es nicht möglich sein, zur Eingabe eines enumerierten Typs eine freie Texteingabe zu ermöglichen, da Typsicherheit nur durch eine Auswahlliste gegeben ist. Der Editor sollte also die Liste der auswählbaren GUI-Elemente in einem solchen Fall auf Auswahllisten begrenzen. XForms bietet dazu zwei Varianten an: `select1` für die exklusive Auswahl von einem aus n Elementen, und `select` für eine beliebige Kombination von m aus n Elementen. Nur das erste von beiden darf hier zum Einsatz kommen.

Formularkomponenten (*form components*) enthalten die einzelnen GUI-Elemente in einem Formular. Neben dem obligatorischen Knopf zum Absenden einer Eingabe sind dies entweder Eingabeelemente und Auswahlelemente für Eingabenachrichten oder Ausgabeelemente für Ergebnisnachrichten. Neben den GUI-Elementen ist auch deren Adressierung in den Formularkomponenten hinterlegt, um eine Zuordnung im Schema zu ermöglichen.

Die Auswahl von Formularkomponenten im Editor für jeden Nachrichtenbestandteil sollte möglich sein. Dabei sind gewisse Grenzen entsprechend der zugrundeliegenden Datentypen zu beachten. Sehr sinnvoll ist eine Auswahl durch den Nutzer bei generischen String-Datentypen, hinter denen sich ein einzeiliges oder mehrzeiliges Eingabefeld verbergen kann, oder aber ein Feld zur Eingabe eines Passwortes. Eine Analyse der Datentypen wurde im Rahmen der Konzeptevaluierung durchgeführt und wird im Anschluss präsentiert.

Umschaltung durch den Nutzer (*user-toggle*) ist ein Konzept, welches von WSGUI-Engines intern verwendet wird, sofern keine Formularkomponente vorliegt. Dieses Konzept benötigt keine Dateiunterstützung, und braucht somit nicht im Editor umgesetzt werden. Der Editor selbst allerdings enthält das Konzept als Teil der Auswahl von Formularkomponenten. Hat sich der Anwender ein entsprechendes GUI-Element gewählt und in eine GUIDD-Datei abgespeichert, so braucht der Aufrufer (z.B. Dynvoker) keine Umschaltung durch den Nutzer mehr anzubieten. Das Konzept der Umschaltung durch den Nutzer führt also zu einer Umgehung des Problems der Ambivalenzen durch fehlende semantische Hinweise.

Seitemweise Darstellung Längere Nachrichten passen auch bei mehrspaltiger Anordnung ab einer gewissen Größe nicht mehr auf den Bildschirm. Das Problem verschärft sich bei kleineren Displays, wie sie auf Subnotebooks, PDAs und Mobiltelefonen anzutreffen sind.

Paginierungsalgorithmen dienen der Aufteilung einer Nachricht in einzeln zu bearbeitende Bestandteile. Dies führt auf embedded-Geräten zu einer Abfolge von Dialogen, im Desktopbereich zu Unterdialogen und im Webbereich zu einer Sequenz von Formularen, wobei Webseiten auch Formulare beliebiger Länge aufnehmen können.

Im WSGUI-Editor wäre es möglich, Paginierungsinformationen mit in den Bearbeitungsvorgang hineinzunehmen. Da die Aufrufanwendungen diese jedoch nicht unterstützen, wurden diese ausgespart. Allgemein sind Paginierungsalgorithmen und andere Methoden zur Verbesserung der Interaktionsmuster vor allem mit mobilen Geräten noch Gegenstand aktueller Nachforschungen, wie [8] belegt.

MIME-Typen Zur Darstellung von Ergebnisdaten ist es wichtig, den Typ der Daten zu kennen. Es kann sich dabei um eingebettete Bilder, URLs oder einfach nur Text handeln. Als allgemeiner Standard für die Angabe solcher Inhaltsmodelle hat sich im Internet das aus dem E-Mail-Bereich stammende MIME (*Multipurpose Internet Message Extensions*) bewährt, welches Werte wie `text/plain`, `application/-pgp-encrypted` oder `image/jpeg` zulässt.

Mit XForms 1.1 sind Ausgabefelder um ein Argument zum Inhalt erweitert worden. Dadurch ist es möglich, bei Editierung einer Ausgabenachricht im Editor den MIME-Typ zu erfragen, und diesen dann als Inhaltsangabe an ein `xforms:output`-Feld anzuhängen, welches in ein Element `guidd:outputComponent` verpackt wird.

Um die Kompatibilität mit XForms 1.0 zu gewährleisten, sollte es möglich sein, die älteren GUIDD-Einträge mit Erwähnung der MIME-Typen als `guidd:outputTypes` vorzunehmen. Die bereits vorgeschlagene Kapselung von abstrakten Formularelementen in einer Bibliothek unterstützt diese Methode der Rückwärtskompatibilität. Die Präferenz für XForms 1.0 oder XForms 1.1 könnte dabei jeweils beim Speichern oder aber global in den Editoreinstellungen abgefragt werden.

Anordnung der Elemente Eine nicht im GUIDD-Umfang enthaltene, aber dennoch potentiell für die Dialoggenerierung wichtige Informationsmenge ist die Anordnung

der einzelnen Elemente. Je nach Anwendungsgebiet sind es entweder Layouts, die ineinander verschachtelt eine hierarchische Anordnung bieten, oder aber absolute und relative Koordinaten der Elemente zu einem Ursprungsfeld wie einem Container zur Gruppierung oder dem Dialogfenster.

Eine Dissertation zu dem Thema [48] präsentiert Algorithmen, die eine möglichst optimale Anordnung von Elementen in abstrakter Art und Weise ermöglichen. Auch dabei sind Möglichkeiten zur Beeinflussung des Algorithmus vorgesehen, denn automatische generierte Layouts sind ebenfalls in den wenigsten Fällen ohne Zusatzinformationen erzeugbar.

Beispiele für problematische Bereiche sind die Anzahl der Spalten, Auslagerung von Elementen in eigene Tabs (Notizbuchseiten), die Reduktion der virtuellen vertikalen optischen Linien durch Unifikation der Elementbreiten. Es tauchen Begriffe wie harmonische, balancierte und symmetrische Fenster auf, die oftmals nur über Berechnung spezieller Optimierungsalgorithmen erzeugbar sind.

Der WSGUI-Editor selbst enthält keine derartigen Algorithmen und dient daher auch nicht zur Erstellung von Informationen zur Anordnung. Es sollte allerdings möglich sein, mit einem WYSIWYG-Editor ein in Bearbeitung befindliches Dokument nachzueditieren und die Positionierungsdaten wieder zu integrieren. Damit ist es möglich, die standardmäßige baumförmige Anordnung der Elemente in den Dialogen manuell zu übergehen.

Es ist vorstellbar, Layoutalgorithmen zukünftig einfließen zu lassen, um bereits die exportierten Daten sinnvoll anzuordnen und somit die Notwendigkeit einer Nachbearbeitung zu minimieren. Eine formatübergreifende Ablage der Daten ist jedoch alles andere als trivial - bereits XForms kann über die Hostformate XHTML und CSS (relative Positionierung) und SVG (absolute Positionierung) nicht einfach damit ausgestattet werden.

HTML-Einbindung Da Formulare selten ohne Kontext existieren, existieren WSGUI-Konzepte zu ihrer Einbindung in ein Rahmendokument. In der Praxis empfiehlt sich dies vor allem für den Webbereich.

So ist es möglich, CSS-Dateien anzugeben, welche die entstehenden Formulare entsprechend vom Darstellungsstil her anpassen. Desweiteren existieren die sogenannten Sektionen. Dies sind markierte Stellen in existierenden (X)HTML-Seiten, die durch das erzeugte Formular ersetzt werden. Mit diesem Template-Mechanismus lassen sich Formulare nahtlos in Webseiten, die bereits einem bestimmten Stil unterliegen, einbinden.

Der WSGUI-Editor soll es ermöglichen, auf Dienstebene entsprechende webspezifische Einstellungen vorzunehmen. Da zum Zeitpunkt der Arbeit die Sektionen nicht Bestandteil des GUIDD-Formates waren, ist dafür ein separater Namespace, aus Effizienzgründen jedoch die gleiche Datei wie für die anderen GUIDD-Informationen zu verwenden.

Datentyp	Häufigkeit
Ganzzahlen (xsd:int)	1761
Große Ganzzahlen (xsd:long)	84
Kleine Ganzzahlen (xsd:short)	16
Fließkommazahlen (xsd:float)	219
Fließkommazahlen mit doppelter Genauigkeit (xsd:double)	1095
Wahrheitswerte (xsd:bool)	766
Binärdaten, base64-kodiert (xsd:base64Binary)	60
Zeichenketten (xsd:string)	14176
URLs (xsd:anyURI)	5
Datums- und Zeitangaben (xsd:dateTime)	216
Unspezifizierter Typ (xsd:anyType)	8

Tabelle 4.2.: Verwendung von XSD-Datentypen in WSDL-Dateien

Der überwiegende Teil der WSGUI-Konzepte braucht also entweder eine Umsetzung im Editor selbst oder muss vom Nutzer durch den Editor umgesetzt werden können. In die erste Gruppe fallen die Konzepte Inferenz und Umschaltung durch den Nutzer, in die zweite Formularkomponenten, MIME-Typen, die Anordnung von GUI-Elementen und die HTML-Einbindung. Das Konzept der seitenweise Darstellung würde ebenfalls in die zweite Gruppe eingeordnet werden, wobei WSGUI-Engines zumindest für eine Ausgabe im XForms-Format durch Inferenz von Paginierungsinformationen aus der Typstruktur dies mit noch nicht bekannter Qualität automatisch durchführen könnten.

4.4.2. Typnutzung in Schemadateien

Da der Einsatz der Formularkomponenten von den Datentypen der assoziierten Elemente und Attribute abhängt (siehe Inferenz), ist es sinnvoll zu wissen, welche Datentypen üblicherweise verwendet werden. Im Rahmen der WSDL-Analyse sind sämtliche in den Dateien enthaltenen Schemata nach den verwendeten Typen untersucht worden. Neben nutzerdefinierten Ableitungen und komplexen Typen werden hauptsächlich die eingebauten XSD-Typen verwendet. Der Häufigkeit nach zu urteilen sind dies vor allem Fließkomma-Typen (1314 Vorkommen), Integer-Typen (1861) und Zeichenketten (14176). Eine Übersicht bietet Tabelle 4.2.

Alle Vorkommen von Strings bieten sich für Formularkomponenten an. Auch Integer-Typen, bei denen der Wertebereich bekannt ist, lassen sich mit Schiebereglern (range) komfortabler und sicherer Eingeben als mit generischen Eingabefeldern. Der Datentyp xsd:base64Binary könnte auf eine hochzuladende Datei hinweisen, während xsd:anyType gar keine Inferenz zulässt und dem Nutzer freie Wahl der Formularkomponente lässt.

Als GUI-Elemente aus dem XForms-Bestand können nunmehr `input` für freie Texteingaben, `secret` für Passworteingaben, `upload` für Datenfelder mit Dateinamen und `textarea` für mehrzeilige Eingaben vom Nutzer des Editors vorgegeben werden. Auswahlelemente wie `select`, `select1` und `range` hingegen erfordern Inferenzmechanismen

im Editor und sollten vom Nutzer nicht auswählbar sein.

Als GUI-Element für XSD-Elemente mit komplexem Typ, welche als Container für andere GUI-Elemente dargestellt werden, wird ebenso wie für die Editierung von Ausgabenachrichten das Element `output` genutzt. Im ersten Fall wird das Element jedoch nur genutzt, um die Beschriftung und einen Hilfetext zu transportieren, das XForms-Element `output` wird für Containerelemente wird nicht angezeigt.

All diese feinen Unterscheidungen in der Anwendbarkeit von GUI-Elementen sollen dem Nutzer verborgen bleiben. Dazu ist eine Bibliothek mit abstrakten Formularelementen denkbar, die XForms-Ausgaben ermöglicht, jedoch mit den gleichen Informationen auch andere Toolkits unterstützt.

4.4.3. Strukturnutzung in Schemadateien

Eine weitere Analyse der in den WSDL-Dateien enthaltenen Schemablöcke wurde auf struktureller Ebene durchgeführt. Tabelle 4.3 enthält die Auswertung auf Zahlenbasis. Der Verwendungsgrad der möglichen Syntaxelemente war dabei höchst unterschiedlich. Einige WSDL-Dateien verwendeten gar keine zusammengesetzten Nachrichtentypen, sondern kommen mit den eingebauten XSD-Typen aus. Der überwiegende Teil jedoch enthielt bis zu fünf miteinander verknüpfte Schemablöcke, welche anwendungsspezifische Datentypen definieren.

Dabei kann man die Gruppe derer, die nur drei bestimmte Strukturelemente enthalten (nämlich `complexType`, `sequence` und `element`) nochmal zusammenfassen. Es sind dies die minimalen komplexen Nachrichtenelemente, die andere komplexe oder einfache Kindelemente besitzen, ohne jedoch bei den einfachen Typen von den XSD-Typen abzuweichen oder weitere XSD-Strukturmerkmale nutzen. Etwa ein Drittel aller WSDL-Dateien geht jedoch darüber hinaus und macht von Ableitungen und Einschränkungen von existierenden einfachen und komplexen Datentypen Gebrauch. Die knappe Hälfte davon, insgesamt also etwa ein Sechstel, beschränkt einfache Typen und ist aus diesem Grund für die WSGUI-Einstellungen besonders interessant. Neben Enumerationen (`enumeration`) werden Muster von Zeichenketten mit regulären Ausdrücken (`pattern`) sowie Intervalle von numerischen Typen als Paar von `minExclusive` und `maxExclusive` verwendet.

Derart eingeschränkte Typen können automatisch durch eine WSGUI-Engine in spezielle Eingabefelder überführt werden (Inferenzkonzept). Gemäß der Analyse wären Auswahllisten, eingebettete Masken für reguläre Ausdrücke und Schieberegler dadurch bereits identifiziert und bräuchten nur noch eine entsprechende Beschriftung durch den WSGUI-Editor erhalten.

Als Beobachtung sei noch vermerkt, dass viele Strukturmerkmale wie `union` oder `choice` überhaupt nicht aufgetreten sind. Auch Attributdefinitionen treten erstaunlicherweise so gut wie gar nicht auf. Da sich jedes XML-Attribut als Element darstellen lässt, der Umkehrschluss aber nicht zutrifft, wäre eine Erklärung für dieses Phänomen gegeben.

Im Gegensatz zu einem Webservice-Aufrufer wie dem Dynvoker wird die Baumdarstellung im WSGUI-Editor eine reduzierte Form eines Schemas benötigen. Der Grund

Strukturmerkmal	WSDL-Dateien mit Verwendung	Anteil
simpleType	73	16,8%
complexType	366	84,1%
element	334	76,8%
attribute	5	1,1%
list	1	0,2%
any	36	8,2%
sequence	342	78,6%
restriction	73	16,8%
enumeration	69	15,9%
minInclusive + maxInclusive	1	0,2%
pattern	3	0,7%

Tabelle 4.3.: Verwendung von XSD-Strukturmerkmalen

dafür ist, dass einige der hier gefundenen Syntaxelemente von XSD-Dateien nur für die Verwendung zur Laufzeit interessant sind. So bietet `choice` eine Wahl zwischen verschiedenen Kindelementen an, von denen dann nur eins ausgewählt werden muss. Im Editor müssen jedoch alle potentiell auftretenden Kindelemente gleichzeitig bearbeitet werden.

4.5. Verwandte Arbeiten

Editoren speziell zur Augmentierung von Webservices mit GUI-Merkmalen gibt es bisher noch nicht. Aufgrund der Verwandtschaft des Themas mit anderen Domänen der modellgetriebenen GUI-Generierung sind aber zumindest andere GUI-Editor-Implementierungen für einen Vergleich gefunden worden und sollen durch zwei Vertreter namens *GUI-Builder* und *JForms* an dieser Stelle kurz vorgestellt werden. Beide Editoren sind im Rahmen von Diplomarbeiten entstanden.

Ein GUI-Builder in Java [46] erzeugt Oberflächenbeschreibungen in einer abstrakten Sprache zur Maskenbeschreibung namens GUIDL (*GUI Description Language*). Zum Einsatz kommt GUIDL mit einer Skriptsprache namens EAL (*Event & Action Language*). Sowohl das Format als auch die Sprache sind firmeneigene Entwicklungen, so dass sie nicht in die Vorbetrachtungen dieser Arbeit eingeflossen sind. Bedeutender sind die praktischen Resultate.

In der Konzeptphase des GUI-Builders sind als WYSIWYG-Werkzeuge Qt Designer und JBuilder evaluiert worden. JBuilder generiert Java-Code und erfüllte damit nicht die Kriterien der Plattformneutralität. Qt Designer erzeugt hingegen XML-Dateien, für die ein XSL-Stylesheet zur Transformation in GUIDL entworfen wurde. Die Korrektheit und Vollständigkeit der Transformation konnte aber ohne erheblichen Mehraufwand nicht erreicht werden. Auch sind die im Qt UI-Format ausdrückbaren Aktionen (Signale und Slots) weniger mächtig als die notwendigen EAL-Konstrukte. Aus diesem Grund ist der GUI-Builder schließlich als Neuentwicklung entstanden, wobei der Qt Designer

für einige Bestandteile Pate stand. Die Arbeit dazu enthält leider keine Betrachtungen zur Anbindung der GUI an das zugrundeliegende Modell. Auch die Betrachtungen zur Internationalisierung beschränken sich auf den Editor selbst und stellen keine Konzepte zur mehrsprachigen Generierung von Oberflächen vor.

Für den WSGUI-Editor wird momentan der Einsatz von Qt Designer als ausreichend betrachtet. Sollten zukünftige WSGUI-Informationen spezielle Angaben zum Layout und zur Geometrie benötigen, die nicht im Qt UI-Format ausdrückbar sind, so ist eine Modifikation oder eine Neuentwicklung nicht ausgeschlossen.

JForms ist ein Hilfseditor im Rahmen einer Anwendung, welche Masken basierend auf Datenbankschemata erzeugt [33]. Auch als Maskengenerator bezeichnet, ähnelt er dem WSGUI-Editor insofern, als dass eine Überführung der Datentypen in den Tabellen der Datenbank in Entsprechungen des GUI-Formats stattfinden muss. JForms erzeugt dabei allerdings die GUI zur Aufrufzeit ohne eine Zwischenstufe wie Java-Code oder einem XML-Format. Die Einstellungen dazu werden in Java-Property-Dateien abgelegt. Ein WYSIWYG-Editor ist als Erweiterung vorgesehen gewesen, wurde aber nicht implementiert. Als Vorlage sollte dazu JavaWorkshop dienen. Die Ausführung der mit JForms erzeugten Masken führte dann auch zu unansehnlichen Layouts, wobei aufgrund der Datenbankstruktur noch nicht einmal Baumstrukturen berücksichtigt werden mussten.

Aufgrund der Fokussierung der Arbeit auf mathematische Grundlagen im Datenbankbereich ist einerseits eine solide Methodik für die GUI-Generierung gewonnen worden, andererseits sind die Möglichkeiten wie verständliche Beschriftungen der Datenfelder leider nicht ausgenutzt worden.

Sowohl der GUI-Builder als auch JForms sind somit nur bedingt als Vorbilder für den WSGUI-Editor geeignet. Dies ist teilweise mangelnden Konzepten geschuldet, teilweise aber auch den unterschiedlichen Einsatzfeldern.

5. Entwicklungsstufen und Merkmale

Der im Rahmen der Arbeit entstandene Editor geht über einen Prototyp hinaus. Es ist eine funktionsfähige Anwendung, die zwar an vielen Stellen noch erweiterbar ist, für die praktische Arbeit allerdings sämtliche Funktionen bereits beherrscht. Abgerundet wird die Software durch eine Dokumentation im Format eines Anwenderhandbuchs, und durch die Möglichkeit der zweisprachigen Bedienung in Deutsch oder Englisch.

5.1. Bestandteile der Software

Die Editor-Software gliedert sich in zwei Hauptkomponenten: den Editor für Diensteigenschaften und den für Nachrichteneigenschaften. Desweiteren wurden, unsichtbar für den Nutzer, Bibliotheken und Klassen für die Verwaltung der unterstützten Dateiformate wie GUIDD, WSDL und Qt UI entwickelt. Schließlich gibt es optional eine Integration in den Qt-Designer mit Hilfe einer XForms-Widget-Sammlung und mit weiteren externen Programmen.

Der Editor wurde so konzipiert, dass er sich dem Nutzer weitestgehend anpassen kann. So lässt sich einerseits die Sprache der Anwendung aus den verfügbaren Übersetzungen auswählen, andererseits auch eine mehrsprachige Editierung aller Einträge aktivieren. Weitere Einstellungen betreffen das Erscheinungsbild des Editors und die Dateiformate, mit denen gearbeitet wird.

Die Programmierung des Editors erfolgte in C++ mit einer resultierenden Codebasis von etwa 6000 Zeilen. Die Abbildung 5.1 vermittelt einen Eindruck von der Aufteilung der Entwicklungsarbeit auf die einzelnen Bestandteile, wobei natürlich die Zeilenanzahl nur eine begrenzt nützliche Metrik für die Aufwandsabschätzung ist [65].

Die Bestandteile werden nachfolgend in eigenen Abschnitten vorgestellt werden.

5.2. Diensteditor

Für die Erzeugung von WSGUI-Informationen für einen Webservice wird dessen WSDL-Datei im Diensteditor geladen und dort entsprechend seiner Bestandteile um GUI-Informationen angereichert. Die Bestandteile sind gemäß der WSDL-Spezifikation Dienste, Operationen, Nachrichten und Nachrichtenteile.

Nachdem eine WSDL-Datei geladen wurde, beginnt eine imaginäre Editor-Sitzung. Diese dauert an, bis entweder eine neue WSDL-Datei geladen oder aber der Editor beendet wird. Innerhalb einer Sitzung lässt sich eine GUIDD-Datei laden, die Editierung kann aber auch ohne eine solche Vorlage beginnen. In beiden Fällen wird das Ergebnis

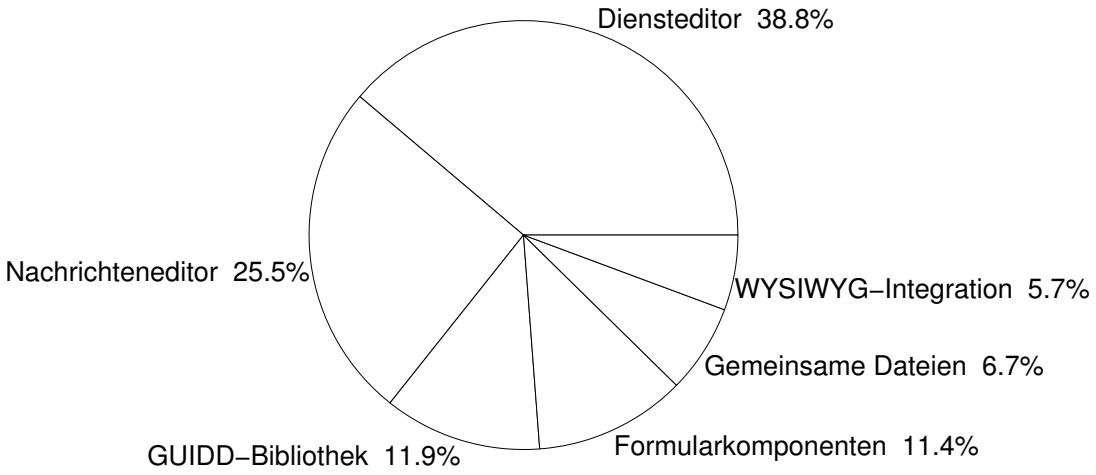

Abbildung 5.1.: Quelltextgrößen der Bestandteile des WSGUI-Editors

anschließend als GUIDD-Datei abgespeichert. Zusätzlich können GUIDD-Dateien auch auf einen Server publiziert werden, um dort von einer WSGUI-Engine geladen zu werden.

Die WSDL-Datei und ihre Bestandteile werden entweder als Baumstruktur oder grafisch-strukturell angezeigt. Die beiden Anzeigen werden synchron mit der WSDL-Datei und der für sie erzeugten WSGUI-Informationen gehalten. Dies entspricht einer Umsetzung des Model-View-Controller-Prinzips (MVC). Die strukturelle Ansicht ist für eine eventuelle spätere Erweiterung um WSGUI-Editierung verknüpfter Dienste ausgelegt, sie kann aber auch je nach persönlicher Präferenz bereits für einen einzigen Dienst genutzt werden.

In der Baumansicht wird bei geladener GUIDD für jeden Eintrag eine Markierung vorgenommen, die auf die Vollständigkeit der GUIDD-Einträge schließen lässt. Grüne, gelbe und rote Markierungen repräsentieren vollständige, teilweise und nicht vorhandene Abdeckung von Dienstmerkmalen mit GUIDD-Informationen. Als weiterer Hinweis wird beispielsweise für eine Operation ihr anzeigbarer Name (`prettyName`) eingeblendet.

Der Editor erlaubt per Kontextmenü sowohl eine Editierung der jeweiligen Bestandteile als auch eine Anzeige (im Falle von Nachrichten) bzw. einen Aufruf (bei Operationen). Dabei wird nur die Editierung von Diensten, Operationen und Fehlernachrichten im Diensteditor selbst vorgenommen, während die Nachrichten zur Ein- und Ausgabe und deren Bestandteile die Domäne des Nachrichteneditors sind, und zur Anzeige bzw. Aufruf auf extern installierte Programme zurückgegriffen wird. In Abbildung 5.2 ist diese Aufgabenteilung (*divide and conquer*) entsprechend dargestellt.

Sofern keine mehrsprachige Editierung gewählt wurde, bietet das Menü Einträge für den Export und Import von Übersetzungskatalogen an. Der Export wurde für drei Formate implementiert: Das aus der GNU-Welt stammende *gettext*, das Qt-eigene Format und die im Java-Umfeld genutzten Property-Dialoge. Der Import ist derzeit auf gettext-Dateien beschränkt. Keines der drei Katalogformate ist wirklich für mehrsprachige Aggregation geeignet – weder Quell- noch Zielsprache können in die Dateien eingetragen werden. Ein Eintrag als Kommentar wäre eine Möglichkeit zur Umgehung des Problems,

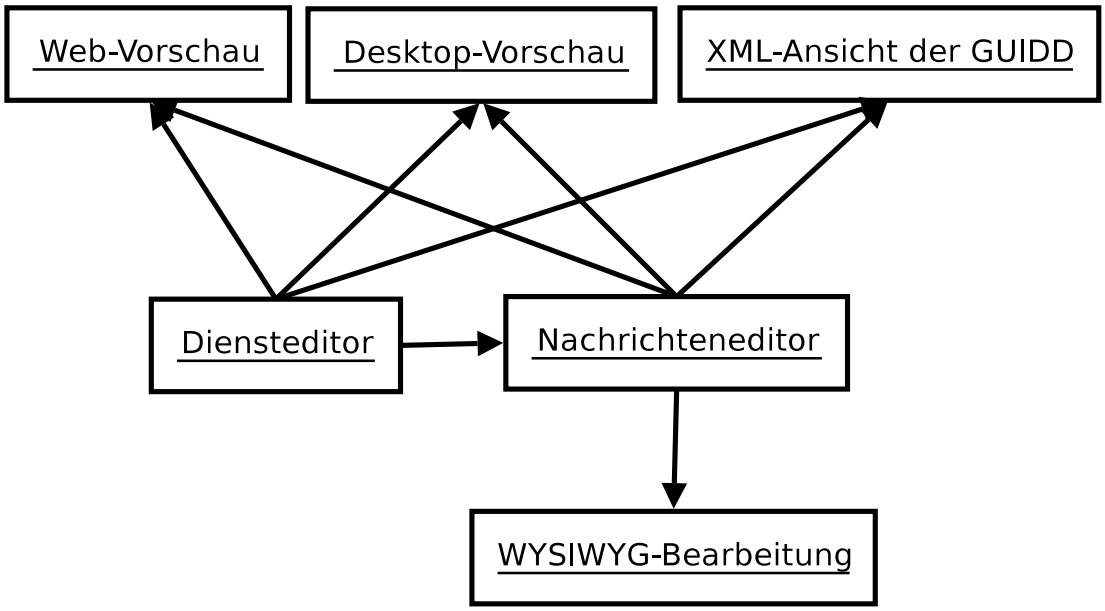

Abbildung 5.2.: Aufrufarchitektur des WSGUI-Editors

würde aber darauf vertrauen müssen, dass Übersetzungswerkzeuge diese Kommentare auch stets an Ort und Stelle lassen. Die Implementierung erfordert nunmehr eine explizite Angabe der Sprache durch den Nutzer zu Beginn der Import- und Exportroutinen.

Über den Diensteditor werden auch die anwendungsweiten Einstellungen vorgenommen. In dem entsprechenden Konfigurationsmenü lassen sich die Sprache des WSGUI-Editors, die bevorzugten Sprachen zur Editierung von GUIDD-Dateien und weitere personengebundene Daten einstellen.

5.3. Nachrichteneditor

Der Nachrichteneditor dient einer XSD-gebundenen Erstellung von WSGUI-Informationen. Die XSD-Daten kommen dabei entweder aus einem allein stehenden Dokument, oder aber aus der Typdefinition einer Nachricht in einer WSDL-Datei.

Neben diesen Grunddaten können WSGUI-Informationen aus einer GUIDD-Datei hinzugeladen werden, was automatisch geschieht, wenn der Nachrichteneditor vom Diensteditor aus aufgerufen wurde. Ähnlich wie im Diensteditor wird eine farbige Markierung sowie eine Anzeige des darstellbaren Elementnamens je nach Abdeckung mit WSGUI-Informationen vorgenommen.

Die Bearbeitung ist sowohl in einer Schema-Ansicht als auch in einer Instanz-Ansicht möglich. Erstere bildet die Definitionen von Elementen einfachen Typs sowie komplexer Typen ab, während letztere auf einer initialen Instanz des Schemas beruht. Die Instanz entspricht dabei in etwa der Ansicht, die der Nutzer beim erstmaligen Aufruf eines erzeugten Dialogs erhält, ohne eventuelle spätere Modifikation durch Hinzufügen von Elementen

in einer Liste.

Es gibt allerdings ein paar Unterschiede, die der Nutzung im Editor geschuldet sind, die sogar dazu führen, dass eine Instanz bearbeitet werden muss, die nicht wirklich dem Schema entspricht. Dies ist insbesondere dann der Fall, wenn von mehreren verschiedenartigen Elementen nur eins in einer gültigen Instanz auftauchen darf oder aber die Häufigkeit des Auftretens speziell eingeschränkt ist. Als Begriff für die Instanz, welche sich darüber hinwegsetzt, wird an dieser Stelle *vollständig-initiale Instanz* vorgeschlagen. Die vollständig-initiale Instanz wird somit nach dem folgenden deterministischen Regelwerk gebildet:

- Ein Element einfachen Typs oder ein Attribut wird in ein XML-Element ohne weitere Kindelemente transformiert.
- Ein Element komplexen Typs wird in ein XML-Element transformiert, und die Bestandteile der Typdefinition (Elemente und Attribute) als dessen Kindelemente entsprechend nach der ersten Regel.
- Gibt die Häufigkeitsbeschränkung eines komplexen Typs eine minimale Häufigkeit eines (einfachen oder komplexen) Bestandteils an, so wird in der initialen Instanz dennoch nur ein XML-Element erzeugt.
- Liegt eine Auswahl von einem aus mehreren Elementen vor, so müssen alle Elemente als XML-Elemente erzeugt werden.

Um diese Transformation zu erläutern, soll das nun folgende Beispiel die Gewinnung der vollständig-initialen Instanz aus einer einfachen Typdefinition demonstrieren, welche in einer fiktiven WSDL-Datei für ein Element (respektive Nachrichtenbestandteil) namens `message` genutzt wird.

```
<xsd:complexType name="Message">
  <xsd:element name="id" type="xsd:int"/>
  <xsd:element name="statements" type="xsd:int" minOccurs="10"
     maxOccurs="10"/>
  <xsd:choice>
    <xsd:element name="priority" type="xsd:string" />
    <xsd:element name="priorityvalue" type="xsd:float" />
  </xsd:choice>
  <xsd:attribute name="credentials" type="xsd:string"/>
</xsd:complexType>
```

Die vollständig-initiale Instanz lautet demnach:

```
<message>
  <id/>
  <statements/>
  <credentials/>
  <priority/>
  <priorityvalue/>
</message>
```

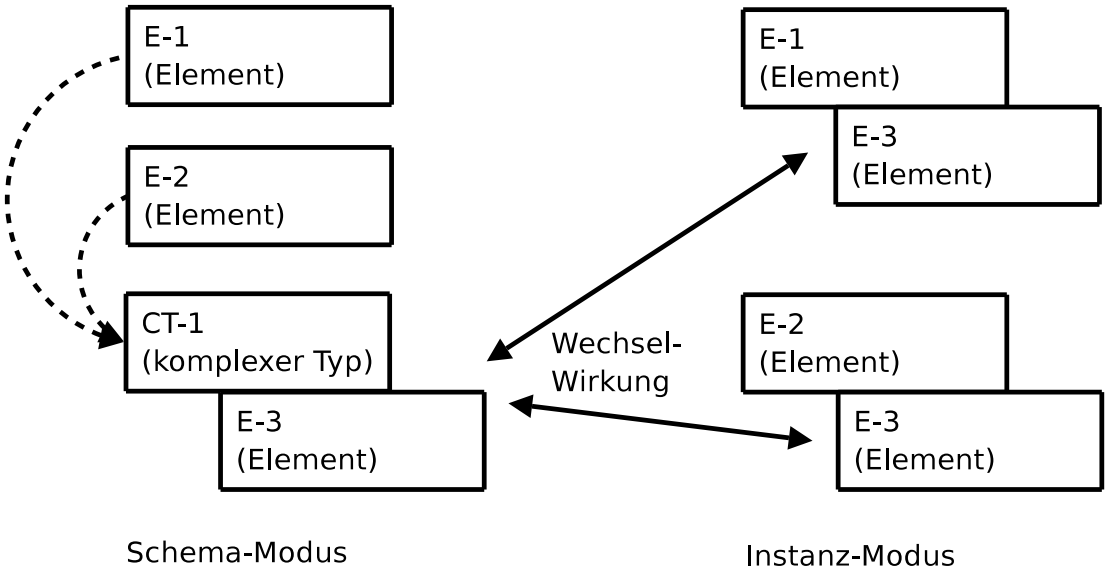

Abbildung 5.3.: Bimodale Nutzung des Nachrichteneditors

Der Unterschied zur erzeugten GUI liegt darin, dass eine WSGUI-Engine das Element `statements` exakt 10 mal anzeigen müsste. Auch die gleichzeitige Verwendung von `priority` und `priorityvalue` wäre nicht möglich. Aber auch zur serialisierten Übertragung der Instanzdaten aus einer GUI, beispielsweise als SOAP-Nachricht, besteht ein Unterschied, da diese das Attribut `credentials` als XML-Attribut enthielten. Terminologien und Übersichten zu den Beziehungen zwischen Schemata und ihren Instanzen gibt es leider nur wenige, hier sei auf [25] verwiesen.

Der Dualität zwischen einem Schema und einer Instanz wird auch in der Adressierung von Formularkomponenten in einer GUIDD-Datei Rechnung getragen. Die beiden Attribute dazu lauten `xpath` für das Schema und `ixpath` für eine gültige oder vollständig-initiale Instanz. Der Nutzer des Editors kann für beide Ansichten Änderungen vornehmen, wird dabei jedoch stets gewarnt, wenn eine Modifikation Auswirkungen auf die jeweils andere Ansicht hat. So ist die Schema-Ansicht ideal, um oft wiederverwendete komplexe Typen nur einmal mit WSGUI-Informationen zu versehen. Soll bei einer bestimmten Verwendung eines solchen Typs jedoch eine Abweichung ermöglicht werden, dann wird durch eine Editierung in der Instanz-Ansicht das editierte Element entkoppelt und erhält einen `ixpath`-Eintrag.

In der Instanzansicht werden nur Elemente und Attribute angezeigt. Die Schemaansicht hingegen enthält auch Typdefinitionen. Da deren Bearbeitung nicht möglich ist, wurde eine Farbkodierung eingeführt, um diesem Umstand Rechnung zu tragen. Elemente und Attribute sind demnach gelb, komplexe und einfache Typen auf der höchsten Ebene (*top-level*) hingegen blau hinterlegt. Je nach Präferenz des Anwenders kann diese Farbgebung abgeschaltet werden. Eine Orientierung mit Hilfe von Tooltips ist in jedem Fall gegeben.

Die Grafik 5.3 stellt die beiden Modi noch einmal gegenüber.

XML-Schema-Eintrag	GUI-Eigenschaften
Element mit komplexem Typ	Name
Attribut sowie Element mit einfachem Typ	Name, Hilfetext, ...
einfacher oder komplexer Typ	(keine)
Gruppe oder Attributgruppe	(keine)

Tabelle 5.1.: Umwandlung von XSD-Elementen in GUI-Elemente

Ein Rechtsklick auf ein Element oder ein Attribut zeigt den Dialog mit den Eigenschaften des zugeordneten Formularelements an. Aus diesen Eigenschaften wird dann die GUIDD-Formularkomponente mit den Attributen `xpath` bzw. `ixpath` sowie dem enthaltenen eigentlichen GUI-Elements erzeugt. Mögliche Angaben umfassen den angezeigten Namen bei allen derartigen GUI-Elementen, die Angabe von Mehrzahligkeit bei Eingabefeldern, und keine weiteren Angaben bei Elementen, welche einen komplexen Typ ohne Inhaltsmodell besitzen, also selbst nur wieder Kindelemente und -attribute enthalten und somit in einer GUI nicht in ein Eingabe-Element, sondern in ein Container-Element umgewandelt werden.

In der Tabelle 5.1 sind die möglichen Eigenschaften noch einmal zusammengefasst.

Je nach Art der editierten Nachricht stehen für Attribute und Elemente mit einfachem Typ entweder Eingabeelemente oder Ausgabeelemente zur Verfügung. Zwar ist die Editierung selbst nicht von XForms abhängig, die Implementierung erzeugt daraus aber XForms-GUI-Elemente. Dies sind `output` für die Ausgabe (und pragmatischerweise auch für Containerelemente, da XForms selbst diese nicht kennt), und `input`, `secret`, `textarea`, `select`, `select1`, `upload` und `range` für die Eingabe.

Innerhalb gewisser Grenzen kann der Nutzer das passende GUI-Element durch eine Sequenz von Entscheidungen aussuchen Abbildung 5.4 zeigt die automatische Vorauswahl durch den Editor und eine Teilmenge der Wahlmöglichkeiten des Nutzers.

Der Nachrichteneditor vereinfacht auch stark die Strukturen von XSD-Dateien. Die ursprüngliche Grammatik enthält viele komplizierte Regeln:

$$
\begin{aligned}
\mathit{Schema} &\longmapsto (\mathit{Element} \mid \mathit{Attribut} \mid \mathit{Gruppe} \mid \mathit{Attributgruppe} \mid \ldots)+ \\
\mathit{KomplexerTyp} &\longmapsto (\mathit{Sequenz} \mid \mathit{Allgruppe} \mid \mathit{Attribut} \mid \mathit{Attributgruppe} \mid \ldots)+ \\
\mathit{Element} &\longmapsto \mathit{EinfacherTyp} \mid \mathit{KomplexerTyp} \mid \mathit{Elementreferenz} \\
\mathit{Attribut} &\longmapsto \mathit{EinfacherTyp} \\
\mathit{Sequenz} &\longmapsto (\mathit{Element} \mid \mathit{Auswahl} \mid \mathit{Gruppe} \mid \ldots)+ \\
\mathit{Auswahl} &\longmapsto (\mathit{Element} \mid \mathit{Gruppe})+ \\
&\ldots
\end{aligned}
$$

Auf Basis der durchgeführten Analyse von Schemadateien wurde entschieden, diese Regeln an zwei Stellen zu vereinfachen: Durch die Auflösung von Redundanzen und durch das Weglassen selten benötigter Merkmale. Letztere würden aber für eine vollständige Unterstützung von XML Schema benötigt werden, so dass auch an dieser Stelle noch eine Verbesserung möglich ist.

Das reduzierte Regelwerk lautet nunmehr:

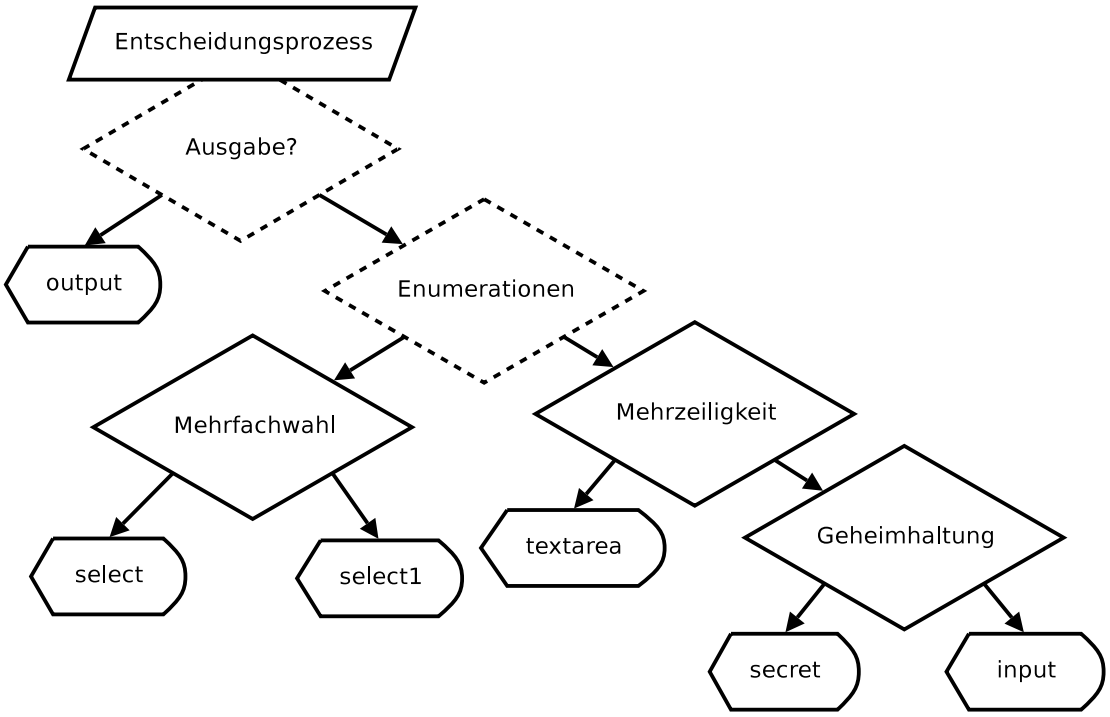

Abbildung 5.4.: Entscheidungsprozess für GUI-Elemente

$$
\begin{aligned}
Schema &\longmapsto Element \mid Attribut \mid EinfacherTyp \mid KomplexerTyp \\
KomplexerTyp &\longmapsto (Element \mid Attribut)+ \\
Element &\longmapsto EinfacherTyp \mid KomplexerTyp \\
Attribut &\longmapsto EinfacherTyp
\end{aligned}
$$

Bezüglich der XSD-Analyse hat sich in der praktischen Arbeit mit dem Nachrichteneditor gezeigt, dass das reduzierte Regelwerk in Verbindung mit dem Algorithmus zur Bildung der vollständig-initialen Instanz für die meisten Webservices ausreichend ist. Es können damit bereits brauchbare Eingabemasken erzeugt werden, ohne dass der Entwicklungsaufwand für diesen Teil des Editors erhöht werden musste.

5.4. Integration von Hilfsprogrammen

Erst durch die Nutzung von externen Programmen wird der Editor vollständig nutzbar. Drei Programme sind dabei gefunden worden, die einen speziellen Bereich abdecken und damit eigene Entwicklungsarbeit am Editor eingespart haben: XML-Editor, WYSIWYG-Editor und Vorschauprogramm.

Eine XML-Ansicht der gerade in Bearbeitung befindlichen GUIDD-Datei lässt sich über einen XML-Editor vornehmen. Dabei werden mehrere Editoren wie beispielsweise `kxmleditor` unterstützt. Sollte keiner dieser Editoren installiert sein, ist eine minimale Ansicht durch eine einfache Editorkomponente durch den WSGUI-Editor vorgesehen.

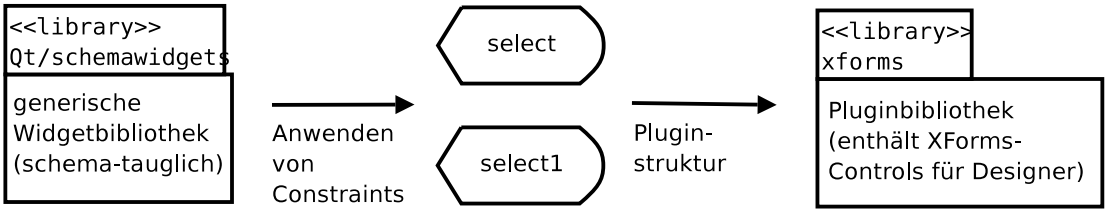

Abbildung 5.5.: Nutzung von Schema-Komponenten im graphischen Editor

Die Auswahl des XML-Editors ist über den Konfigurationsdialog einstellbar.

Eine Ausrichtung von Formularkomponenten anhand eines Layouts oder von Koordination ist mit dem WYSIWYG-Werkzeug Qt Designer möglich. Dabei wird eine deklarative Beschreibung der GUI-Elemente in XML in eine temporäre Datei ausgegeben und diese dann mit dem Qt Designer geladen. Nach der interaktiven Ausrichtung der Elemente speichert der Nutzer die geänderte Datei wieder ab und beendet den Qt Designer. Der WSGUI-Editor kann anschließend die Positionierungsdaten einlesen. Diese Daten werden momentan verworfen, eine zukünftige Nutzung durch WSGUI-Engines könnte der Editor durch Abspeichern in einem definierten Dateiformat unterstützen.

Da Qt Designer selbst nur die Qt-eigenen GUI-Elemente unterstützt und nicht solche mit der Semantik von XForms, die beispielsweise stets ihre eigene Beschriftung mitbringen, wurde eigens für den Export eine Widget-Bibliothek mit XForms-GUI-Elementen geschrieben. In der Abbildung 5.5 ist deren Nutzung verdeutlicht.

Der Aufruf einer WSGUI-Engine ist schließlich unter dem Punkt „Vorschau" integriert. Sowohl die Desktopanwendung Kung als auch das Servlet Dynvoker können dafür genutzt werden. Kung wurde eigens dafür im Rahmen dieser Arbeit mit einer Unterstützung für das GUIDD-Format ausgestattet.

5.5. Autorenprozess

Nach der Vorstellung der Software soll nunmehr gezeigt werden, auf welche Art und Weise der Editor sich in existierende Infrastrukturen im Bereich Webservices einbinden lässt. Dazu wird in die integrierte und die separierte Nutzung unterschieden.

Eine integrierte Nutzung ist verzahnt mit dem Entwurf eines Webservice. Dabei ermöglicht der Dienstanbieter aktiv einen dynamischen Aufruf seiner Dienste, indem er die anderweitig nur maschinell auswertbaren WSDL-Dateien um zumindest in einer Sprache verfasste GUIDD-Dateien ergänzt. Der Verzeichnisdienst, welcher die Webservices verwaltet, z.B. UDDI oder LDAP, bekommt einen zusätzlichen Eintrag für jede zu einer WSDL-Datei korrespondierende GUIDD-Datei. Günstig sind hierfür Verzeichnisdienste, welche beliebige Metadaten zu ihren Einträgen hinzufügen können. UDDI an sich erlaubt das nicht in aller Freiheit, es gibt aber Erweiterungen mit Metadaten-Archivierung [1].

Die separierte Nutzung ist hingegen die nützlichere Variante für bereits existierende Dienste oder solche, deren Anbieter kein Interesse an einer Anreicherung mit GUI-Informationen zeigt. Dabei kann ein Anwender nach Bedarf GUIDD-Informationen er-

stellen und diese lokal vorhalten. Eine WSGUI-Engine, die entweder auf dem Desktop des Anwenders läuft oder als Serveranwendung auf eine publizierte GUIDD-Datei zugreifen kann, kann somit ohne Mitwirkung des Dienstanbieters eine ad-hoc-Nutzung des Dienstes ermöglichen.

Übersetzungen der GUIDD-Informationen in andere Sprachen sind in beiden Szenarien möglich. Eine Optimierung durch den Einsatz des Editors besteht darin, dass sämtliche angezeigte Zeichenketten in ein unter Übersetzern gängiges Format exportiert werden kann, und somit die Tätigkeit der Übersetzung kein Fachwissen zu Webservices verlangt. Nach der Übersetzung werden die bearbeiteten Dateien eingelesen und die übersetzten Zeichenketten in die GUIDD-Datei integriert.

Auch in der Arbeit über den MDA-Editor [70] war ein Export und Reimport von Übersetzungsdateien vorgesehen. Für die Zukunft ist es sicherlich denkbar, Übersetzungsdienste wie Rosetta in eine service-orientierte Architektur einzubinden und vom Editor aus nutzbar zu machen, zumal man die Vorteile automatisch generierter Übersetzungen in steigendem Maße nutzen könnte [54]. Speziell der Bereich der Formularübersetzung, und damit der Übersetzung von Zeichenketten in GUIDD-Dateien, ist dabei schon weit fortgeschritten [10].

5.6. Software-Architektur

Abschließend zur Implementierung informiert dieser Abschnitt über die Umsetzung des Editorkonzepts auf Ebene der Programmierung.

Die Klassenbibliothek Qt wurde um einige C++-Klassen erweitert, um dem Bedarf eines internationalisierten Editors gerecht zu werden. Diese Erweiterungen sind dabei auch für andere Programmiersprachen und Toolkits denkbar, da sie gewissermaßen existierende Paradigmen in der Programmierung übergehen.

Eine solche Erweiterung ist die Klasse `MultiString`. Diese ähnelt einer beliebigen String-Klasse, kann jedoch mehrere Zeichenketten pro Objekt speichern, wobei immer nur eine davon aktiv ist. Mit Hilfe der Klasse lassen sich Übersetzungen sowohl in der GUI als auch in XML-Dateien konsistent durch sämtliche Bereiche der Anwendung handhaben.

Eine Abbildung von XForms-GUI-Elementen wurde durch eine Abstraktionsschicht in der editor-eigenen Bibliothek `forms` geschaffen. Diese Bibliothek ist in Abbildung 5.6 zu sehen. Objekte, die aus Klassen dieser Bibliothek instantiiert werden, können einerseits generisch konfiguriert werden (beispielsweise zum Setzen der Beschriftung), haben andererseits aber je nach GUI-Element verschiedene Einstellungen. Die Serialisierung aller der Objekte in XML-Daten wird für die Einbindung in die GUIDD-Dateien vorgenommen. Formularkomponenten der GUIDD-Bibliothek enthalten folglich keine konkreten XForms-Elemente, sondern Objekte der Forms-Bibliothek, die neben XForms auch andere Darstellungen annehmen könnten. Bereits jetzt sind XForms 1.0 und 1.1 implementiert, die sich leicht in der Repräsentation der GUI-Elemente unterscheiden.

Die `forms`-Bibliothek ist nicht mit der XForms-Unterstützung für den Qt Designer zu verwechseln - erstere ist eine Abstraktionsschicht, letztere beinhaltet konkrete Widgets für die Einbindung in den WYSIWYG-Editor.

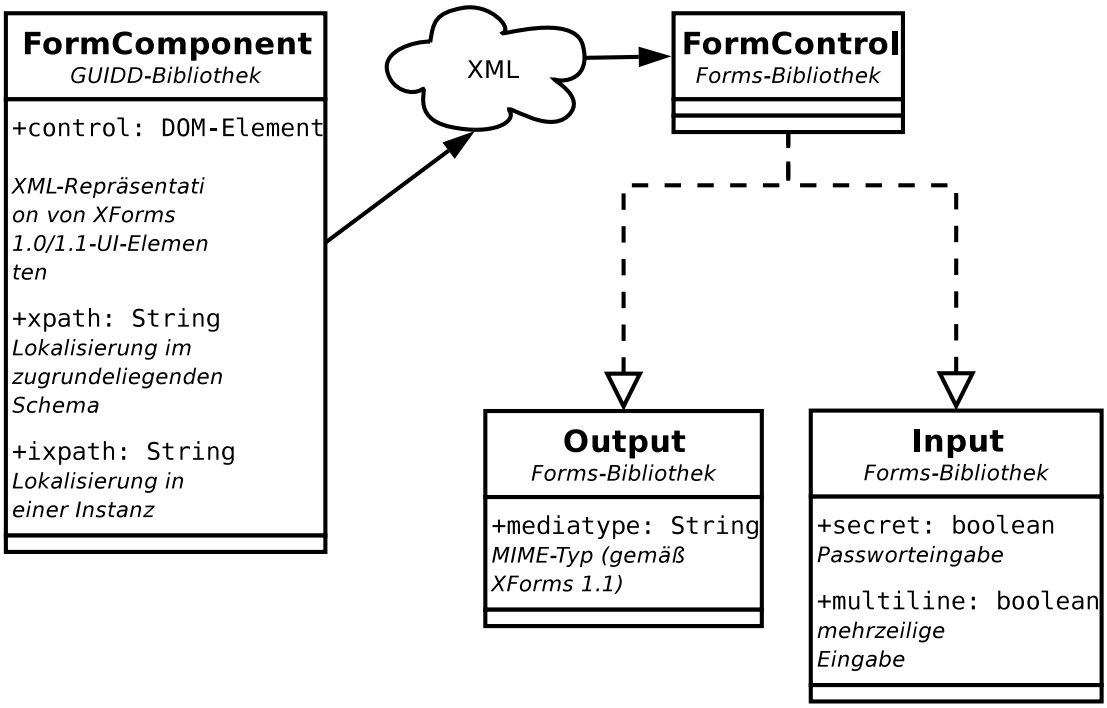

Abbildung 5.6.: Abstrahierte Formular-Elemente auf XForms-Basis

Die Nutzung von netzwerktransparentem Dateizugriff aus dem Editor heraus kann ebenso eingestellt werden wie nutzerfreundlichere Dateidialoge und die automatische Erkennung von Umgebungseigenschaften wie der präferierten Sprache des Anwenders. Dazu ist eine Compilierung unter Einbeziehung von KDE-Bibliotheken notwendig. Sind diese nicht verfügbar, so ist der Editor zwar weniger komfortabel, aber immer noch nutzbar. An dieser Stelle ist zwischen einer allein stehenden Anwendung und einer desktopintegrierten Anwendung zu unterscheiden.

Da neben WSGUI-Konzepten auch Editorkonzepte für einen WSGUI-Editor umgesetzt werden sollten, wurde darauf, sofern möglich, geachtet. Diese Konzepte unterscheiden Editoren von anderen Anwendungskategorien. Eine Auswahl der Konzepte lautet:

- Alle Menüeinträge sind mit Hinweistexten für die Statusleiste versehen.
- Nichtdestruktive Dateiverarbeitung: Bevor eine neue Datei geladen wird, ist bei aktiven Änderungen vom Nutzer eine Bestätigung einzuholen.
- Der Editor kann mit Angabe der zu bearbeitenden Datei als Parameter aufgerufen werden.

Das Konzept der direkt-manipulativen Arbeitsweise wurde durch Kontextmenüs umgesetzt. Andere Editorkonzepte wie ein navigierbarer Editierverlauf (*undo/redo*) und Integration von Werkzeugen wie einer Rechtschreibprüfung wären denkbar, sind aber im Rahmen dieser Arbeit nicht implementiert worden.

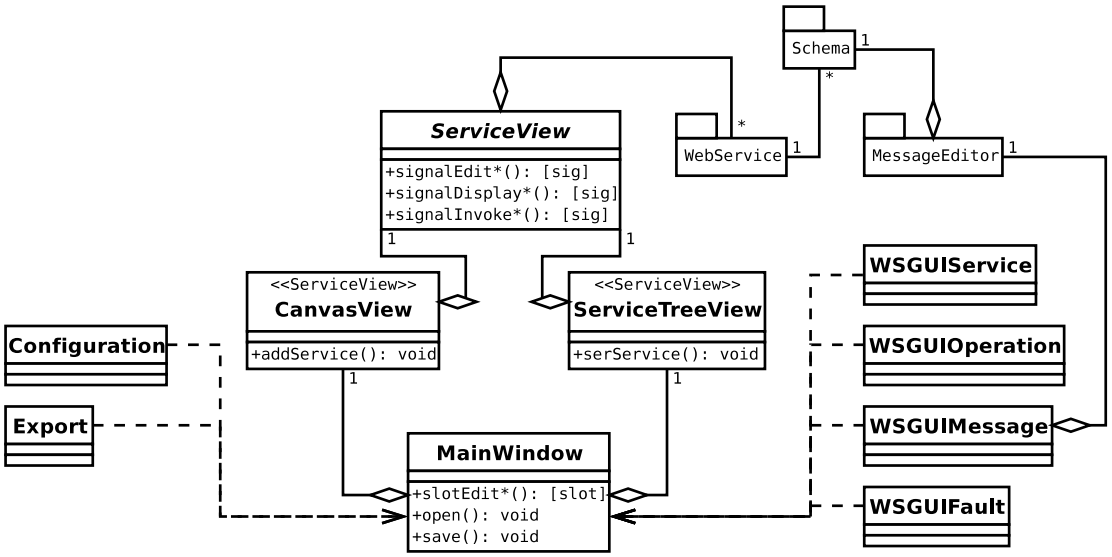

Abbildung 5.7.: Klassenstruktur des Diensteditors

Der Aufbau des Diensteditors auf Klassenebene ist zum Abschluss der Betrachtungen zur Software-Architektur in Abbildung 5.7 dargestellt.

6. Anwendungsgebiete

Die Nutzung von XML-Dateien im Editor ermöglicht eine anwendungsunabhängige Arbeit mit GUIDD-Dateien auf Basis beliebiger Referenzdateien, welche Typdefinitionen im Format XML Schema enthalten. Für den Anwender ist hingegen interessant, wofür die so erzeugten GUIDD-Dateien verwendet werden können und wie die konkrete Einbindung in die jeweiligen Ablaufumgebungen stattfindet. Exemplarisch werden der Trivialfall, also die Nutzung im Kontext von Webservices, sowie die Generierung von Dialogen im Desktopbereich erläutert.

6.1. Webservices

Die Ad-hoc-Nutzung von Diensten ist ein immer wieder gewollter Ansatz zur Einsparung der Entwicklung angepasster Clientprogramme. Eine späte, dynamische Bindung einer generischen Anwendung an jeden beliebigen Dienst ist mit Hilfe von Webservices prinzipiell möglich. Die Nutzbarkeit durch den Menschen erfordert dabei allerdings Hinweise zum Nachrichtenaufbau, die in Dienstbeschreibungsformaten wie WSDL nur formal enthalten sind.

Mit Hilfe des WSGUI-Editors können nun entweder die Entwickler eines Dienstes oder aber jeder beliebige Nutzer eigene angepasste Informationen zur Generierung der Dialoge zur Kommunikation mit dem Dienst erstellen.

Eine WSGUI-Engine, die GUIDD-Dateien verarbeiten kann, ist der Dynvoker [75]. Die Software wurde im Jahr 2006 am Lehrstuhl für Rechnernetze an der TU Dresden entwickelt und erzeugt XForms-basierte Webseiten zur dynamischen Nutzung von Webservices.

Neben der reinen Nutzung des Dynvokers durch beliebige Anwender dient er auch den Autoren von GUIDD-Dokumenten als Testanwendung. So sind Dynvoker-Aufrufe vom WSGUI-Editor aus möglich. Dabei müssen die jeweils lokal bearbeiteten Dokumente vor dem Test auf einen Server geladen werden. Die Konfiguration des Editors erlaubt hierfür die Angabe eines Speicherplatzes, beispielsweise auf einem FTP-Server.

6.2. Konfigurationsdialoge

Der Beginn schemabasierter Oberflächengenerierung im Kode-Projekt entsprang der Idee, die ohnehin schon XML-basierte Verwaltung von Konfigurationsdateien weiter zu vereinfachen und die Dialoge zur Änderung von Anwendungskonfigurationen mit Hilfe eines Schemas und eventueller Hilfestellungen zur GUI-Generierung automatisch zu erzeugen.

Zur Umsetzung der Idee wurde das Subprojekt KXForms [68] gegründet. KXForms kann dabei durch folgende Selbstbeschreibung charakterisiert werden:

> KXForms is a format to describe user interfaces for editing XML data in an abstract way. It comes with an application that creates functional graphical user interfaces from KXForms descriptions. The format is suitable to be generated automatically based on available schema information.

Mit dem Werkzeug `schema2kxforms` ist es möglich, KXForms-Beschreibungen automatisch aus XML Schema zu generieren. Es werden dabei wenige Inferenzmechanismen genutzt, eine Übertragung aller möglichen Methoden [72] könnte die Qualität dieser Anwendung aufwerten. Gleichzeitig beinhaltet das Werkzeug heuristische Methoden, nach denen etwa Beschriftungsfelder im Plural erscheinen, sofern die Häufigkeit eines Elements größer eins ist. Davon wiederum könnte der Dynvoker profitieren, wobei man zur Unterscheidung der deterministischen und der heuristischen Mechanismen als Nutzer stets letztere abschalten können sollte.

Gerade für Serveranwendungen, die heutzutage oftmals eine eigene webbasierte Oberfläche zur Konfiguration mitbringen, könnte man mit Hilfe des Editors GUI-Informationen ablegen, die eine Reintegration der Dialoge in eine Desktopanwendung erlauben. Dies ermöglicht es in vielen Fällen, einen HTTP-Port einzusparen und somit auch die Sicherheit des Servers zu erhöhen, da die generierten Dialoge auf der Desktopseite die Instanz der Konfiguration als Datei über eine einfache Kopieroperationen, beispielsweise unter Nutzung von SSH, übertragen könnten.

KXForms nutzt die GUI-Element-Definitionen aus der XForms-Spezifikation, ohne jedoch eine vollständige XForms-Implementierung darzustellen. Stattdessen werden die Elemente ähnlich wie in einer GUIDD-Datei als Formularkomponenten für bestimmte Knoten des XML-Schema-Baumes abgelegt. Der folgende Ausschnitt aus einer Schema-Datei mit solchen UGHs (*UI Generation Hints*) illustriert die Nutzung.

```
<xs:element name="category">
  <xs:annotation>
    <xs:appinfo>
      <ugh:hint ref="feature">
        <ugh:label>Feature: <arg ref="summary"
            truncate="50"/></ugh:label>
      </ugh:hint>
    </xs:appinfo>
  </xs:annotation>
  <xs:complexType>
  ...
  </xs:complexType>
</xs:element>
```

Die UGHs können dabei also jeweils im `annotation`-Eintrag von XSD als `appinfo` abgelegt werden, da derartige Einträge anwendungsspezifisch sind. Die Ablage als externe UGH-Datei ist jedoch ebenfalls möglich.

Da einer der Vorteile des Einsatzes des WSGUI-Editors die Unabhängigkeit von den GUI-Vorgaben eines Dienstanbieters ist, wäre die Einbindung von GUIDD-Hinweisen in `appinfo`-Einträge zwar für eine zukünftige Revision denkbar, extern bereitgestellte GUIDD-Hinweise sollten dann allerdings im Konfliktfall stets eine höhere Priorität enthalten.

Ein interessantes Konzept ist der Bezug der Beschriftungen (Label) auf Instanzwerte. Somit können Beschriftungen dynamisch an die gerade editierten Daten angepasst werden. Eine Unterstützung für dieses Konzept im WSGUI-Editor müsste dafür implementiert werden.

6.3. Druckdialoge

Die Arbeitsgruppe *Open Printing* der Free Standards Group (FSG) arbeitet intensiv an der Möglichkeit, Dialoge zum Ansteuern von Druckern abstrakt beschreiben zu können. Im Rahmen der Diplomarbeit wurde die Chance genutzt, einzelnen Mitgliedern des Konsortiums die WSGUI-Konzepte nahezubringen und auf fachlicher Ebene einzelne Punkte zu diskutieren.

Drucker werden mit Hilfe sogenannter PPD-Dateien beschrieben (*Postscript Printer Definition*). Diese beinhalten eine Art Schema, allerdings auch programmatische Bestandteile in der Postscript-eigenen Programmiersprache. Diese Durchmischung von deklarativen und imperativen Bestandteilen war nicht förderlich für eine GUI-Generierung. Im Jahr 1998 begann zur Behebung dieses und anderer Probleme eine Initiative namens UPDF (*Universal Printer Description File*) [66]. Urheber war die Printer Working Group (PWG) der Organisation IEEE-ISTO.

Dateien mit UPDF-Informationen basieren auf XML Schema. Diese werden sowohl im permanenten Speicher des Druckers als auch online bereitgestellt, um von einer generischen Treiberkomponente mit Betriebssystem- und GUI-Zugriff genutzt zu werden. Neben etlichen Einstellungen zu Merkmalen der Drucker wie der Papiergröße oder der Auflösung sind auch GUI-Merkmale wie angezeigte Zeichenketten Teil der UPDF-Spezifikation. Desweiteren ist es ähnlich wie im WSGUI-Ablauf möglich, unabhängig vom Anbieter eine eigene lokale GUI-Definition bereit zu halten. Im Umfeld der Druckeradministration befasst sich allerdings nicht der Anwender damit, sondern der Systemadministrator.

Die UPDF-Spezifikation erscheint ausgereift, dennoch fehlt es an einer Implementierung, die aufgrund der Komplexität des Schemas nicht einfach zu erstellen sein wird. Der Dominoeffekt der fehlenden Implementierung ist die weitere Fragmentierung der existierenden Treiberentwicklung, aufgeteilt nach Betriebssystem und grafischer Bedienoberfläche. Nicht zuletzt daher kam im Jahr 2005 abermals die Idee auf, eine Standardisierung voranzutreiben. Ob UPDF dabei eine große Rolle spielen wird, bleibt noch abzuwarten.

Mit WSGUI-Konzepten wäre es möglich, zumindest den Teil der GUI-Generierung beizusteuern. Die Instanzdaten des Druckdialogs würden dann nicht an einen Webservice gesendet werden, sondern an einen lokalen Druckserver. Zukünftig sind sicherlich auch SOAP-Schnittstellen für diese denkbar, so dass sogar mit existierenden Aufrufwerkzeugen wie dem Dynvoker oder Kung die Druckverarbeitung bedient werden kann.

7. Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse der Untersuchungen und der Nutzen des Editors besprochen. Diese Aussagen werden ergänzt durch Hinweise auf zukünftige Änderungen in XML-basierten Standards im Webservice-Bereich.

7.1. Resultat der Arbeit

Es ist ein zweistufiger Editor entstanden, der aufgrund der intensiven Integration bereits existierender Software mit recht wenigen Zeilen Quelltext eine gute Einbindung in die Arbeitsabläufe der WSGUI-Erstellung ermöglicht. Im Gegensatz zu anderen GUI-Editoren wird die Anbindung an die Elemente, Attribute und Datentypen der Nachrichten von Webservices zur Sicherstellung der Struktur- und Typsicherheit gewährleistet.

Im Laufe der Konzeptionierung und Evaluierung von Basistechnologien wurden dabei oft Festlegungen getroffen, die an anderer Stelle anders lauten könnten. So könnte man den Einsatz von RELAX NG-Schemata forcieren, den Editor als Eclipse-Plugin schreiben und im Ausgabeformat Formularkomponenten in XUL ablegen. All dies ist im Rahmen der Arbeit nicht geschehen. Diese Menge an getroffenen Entscheidungen macht deutlich, dass es noch keine allgemein akzeptierte Referenzarchitektur für service-orientierte Architekturen gibt. Oft konkurrieren Spezifikationen verschiedener Gremien um ähnliche Sachverhalte, so dass eine Festlegung auf jeweils eine Spezifikation immer Probleme mit der Interoperabilität in der Praxis aufweisen wird. Eine Unterstützung aller Formate hingegen ist schlichtweg nicht möglich.

Unter diesem Blickwinkel deckt der WSGUI-Editor immerhin einen Großteil der verbreiteten Formate ab. Sobald die Spezifikationen revidiert werden oder neue hinzukommen, wird sicherlich eine Überarbeitung des Editors notwendig werden.

7.1.1. Vorteile durch WSGUI-Editor

Der WSGUI-Editor ermöglicht es, mit wenig Fachkenntnissen in den Bereichen Webservices und XML unterstützende Hinweise für die GUI-Generierung im Rahmen der Interaktion eines Menschen mit einem Webservice zu erstellen. Die erzielte Auslagerung der Übersetzungsvorgänge führt zu einer auf die Aufgabenteilung in Firmen und Organisationen angepassten Arbeitweise. Durch die Einbindung einer Vorschaukomponente ist eine zeitnahe Kontrolle der erstellten Dateien direkt während des Editierprozesses möglich geworden. Da im Gegensatz zu traditionellem GUI-Design im Internet (Webdesign) oder auf dem Desktop (Anwendungserstellung mit WYSIWYG-Editoren) bei automatisch erzeugten GUIs ein modellorientierter Ansatz anstatt eines visuell-orientierten zum

Einsatz kommt, ist dies eine wichtige Komponente, wie auch andere Arbeiten auf dem Gebiet der *single-authored user interfaces* belegen [71].

Eine Einbindung in eine SOA ist je nach Einsatzplattform durch beispielsweise netzwerktransparenten Zugriff auf WSDL- und GUIDD-Dateien ansatzweise realisiert worden.

Eine Ausbaufähigkeit des WSGUI-Editors ist an den folgenden Punkten gegeben:

- Unterstützung des UGH-Formats für die Ausgabe
- Editierung von komplexen Diensten über die grafische Dienstansicht
- Weitere Import-Formate für Übersetzungskataloge
- Editorkonzepte wie *undo/redo*, Rechtschreibprüfung, ...
- Unterstützung aller XSD-Merkmale über die reduzierte Form hinaus

7.1.2. GUIDD-Implementierung

Desweiteren wurde eine GUIDD-Bibliothek für die Programmiersprache C++ erschaffen, welche das für die Umsetzung von WSGUI-Konzepten wichtige Format lesen, schreiben und in einer für die Programmiersprache nativen Art und Weise verarbeiten kann. Neben der Java-Implementierung GUIDD4J ist dies bereits die zweite Bibliothek für dieses Format.

Die von der *Internet Engineering Task Force (IETF)* herausgegebene RFC-Reihe fordert zwei unabhängige und und zusammenarbeitende Implementierungen für alle Spezifikationen, welche auf dem Weg zum Internetstandard die Wertung *draft standard* bekommen sollen, kodifiziert in RFC 2026/BCP 9 [12]. Das XDR-Protokoll, das MIME-Format und das Unicode-Format UTF-8 sind Vertreter derartiger Spezifikationen, so dass der zusätzliche Implementierungsaufwand für die GUIDD-Bibliothek in C++ vor dem Hintergrund, dieses Format einmal formell zu einer Standardisierung einzureichen, als gerechtfertigt angesehen werden kann.

7.2. Verbesserungsvorschläge

Im Laufe der Arbeit sind beim Umgang mit einigen relevanten Spezifikationen verbesserungswürdige Festlegungen erkannt worden. Einige davon sind bereits im Text erwähnt worden, andere haben eher generellen Charakter. Es werden Vorschläge gegeben, um die Spezifikationen entsprechend zu vereinfachen, ohne ihre Ausdruckskraft einzuschränken, oder sie so zu erweitern, dass sinnvollere Szenarien für ihren Einsatz ohne übertriebenen Mehraufwand möglich werden. Die Vorschläge könnten somit in zukünftige Überarbeitungen mit einfließen.

7.2.1. XML Schema

Um die recht komplizierten Beziehungen zwischen einfachen und komplexen Typen mit einfachen oder komplexen Inhalten und leeren Inhaltsmodellen zu vereinfachen, wird vorgeschlagen, den Definitionen eines komplexen Typs das Element `xsd:content` beizufügen.

```
<xsd:complexType name="KomplexerTyp">
  <xsd:attribute name="attr" type="xsd:string"/>
  <xsd:element name="elem" type="xsd:int"/>
  <xsd:content type="my:AndererKomplexerTyp"/>
</xsd:complexType>
```

Existiert das Element `xsd:content`, so ist der komplexe Typ von einem einfachen Typ abgeleitet. Existiert es hingegen nicht, so ist das Inhaltsmodell leer, und Elemente des komplexen Typs enthalten nur Elemente und Attribute. Dabei schließen sich `xsd:element` und `xsd:content` gegenseitig aus.

Eine weitere wünschenswerte Verbesserung von XML Schema ist die zwingende Angabe der URL einer Schema-Datei, wenn diese eingebunden wird. Die `import`-Anweisung benötigt momentan den Namespace und nimmt optional die besagte URL mit auf (als Attribut `schemaLocation`). Dies führt zu Problemen bei WSGUI-Engines, welche den angegebenen Namensraum nicht kennen. Durch eine zwingende Angabe könnte die referenzierte Datei inspiziert werden, während die Programme, die den Inhalt des Namensraumes schon kennen, die URL einfach ignorieren könnten. Vermutlich wurde die Optionalität der URL mit der Idee von nicht-öffentlichen herbeigeführt, die Analyse der WSDL-Dateien in dieser Arbeit hat jedoch gezeigt, dass auch im Internet davon Gebrauch gemacht wird.

Ob XML Schema auch in Zukunft die heutige Rolle spielen wird, ist bei der Menge an Alternativen wie RELAX NG, Schematron und anderen Sprachen zweifelhaft. Allgemeine Vorschläge für Verbesserungen in Schemasprachen und eine Übersicht finden sich in [52].

7.2.2. XForms

Bereits angesprochen wurde die Suboptimalität der Einführung neuer Datentypen als Ersatz für `xsd:duration`. Wenn man bedenkt, dass beispielsweise ein Astronom mit `xforms:yearMonthDuration` Angaben über Jahrtausende in Monaten angeben müsste, andererseits Messergebnisse von einem Sportereignis mit `xforms:dayTimeDuration` einen viel zu umfangreich bemessenen Datentyp nutzen müssten, erkennt man den Anachronismus von zu speziell festgelegten Grenzen innerhalb einer physikalischen Einheit.

Der Vorschlag zur Lösung bezüglich XForms ist die Beibehaltung des Schema-Datentyps `xsd:duration` und einer Angabe von `xforms:preferred` als Attribut, welches die bevorzugte Basiseinheit bezeichnet. Diese Angabe könnte als `xsd:preferred` auch in den Schema-Standard übernommen werden, und stünde somit WSGUI-Inferenz-Mechanismen zur Verfügung.

Sehr positiv ist die Einführung des Attributs `mediatype` für Ausgabeelemente in XForms 1.1 zu bewerten. Eine zwingende Analyse der Inhalte auf ihren MIME-Typ, wie momen-

tan in Kung implementiert, entfällt dadurch.

Weitere Verbesserungen könnten durch die Einführung weiterer Standard-Widgets zur numerischen Eingabe erzielt werden. Speziell Fließkommazahlen unterscheiden sich recht erheblich in der Angabe des Dezimalstellentrennzeichens, so dass eine unterstützende Eingabemöglichkeit die Anzahl der Eingabefehler weiter dezimieren würde. Zwar ermutigt XForms die automatische Auswahl eines GUI-Elements durch den XForms-Prozessor basierend auf dem an das GUI-Element gebundenen Datentyp, auch wenn `xforms:input` angegeben ist, doch führt ein fehlender Ansatz zur Standardisierung der GUI-Elemente über Implementierungsgrenzen hinweg zu wechselhaften Erfahrungen beim Anwender.

7.2.3. WSDL

Dokumentations-Elemente sollten ein Attribut tragen, welches erkennen lässt, ob es sich bei ihrem Inhalt um Fließtext oder um einen HTML-Ausschnitt handelt. Im letzteren Fall wäre die Angabe der HTML-Version obligatorisch.

Einige Veränderungen wird die Version 2 von WSDL bringen, die sich gerade im Standardisierungsprozess befindet [15]. Eine genauere Angabe der Dokumentation zur automatisierbaren Einbindung in Anwendungen ist nach wie vor nicht enthalten. Eine Aufteilung in lesbaren Text und `appinfo` in Ähnlichkeit zu XML Schema würde es ermöglichen, sämtliche Angaben aus einer GUIDD-Datei optional innerhalb einer WSDL-Datei zu lagern.

7.2.4. GUIDD

Das GUIDD-Format sieht momentan keine Aufnahme semantischer Informationen vor. Sicherlich könnte man derartige Informationen auch innerhalb von XML Schema als anwendungsspezifische Hinweise unterbringen, etwa nach dem folgenden Muster:

```
<xsd:element name="id" type="xsd:string">
  <xsd:annotation>
    <xsd:appinfo>
      <sem:hint content="password"/>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Ein Nachteil wäre allerdings, dass es keinen allgemein akzeptierten Standard für derartige Annotationen gibt, und sowohl bereits existierende als auch zukünftige WSDL-Dateien diese wohl nicht enthalten werden. Mit einer Aufnahme in das GUIDD-Format könnte jeder Nutzer des Editors diese Hinweise hinzufügen, ohne dass der Anbieter des Dienstes dazu verständigt werden müsste.

Semantische Erweiterungen für Webservices sind unter dem Namen WSDL-S in Entwicklung und werden sicherlich ebenfalls eine Rolle für die zukünftige Formatierung derartiger Hinweise spielen [63].

Ähnlich den UGHs würden sich auch Formularkomponenten innerhalb der Annotationen von XSD unterbringen lassen. Da nicht für jede Plattform eine XForms-Implementierung existiert, könnten diese abstrakt in UIML-Syntax formuliert werden. Bisher gibt es

allerdings auch keine Abbildung von UIML auf XForms, und viele Interaktionselemente wie die dynamische Größenänderung einer Liste von Eingabeelementen wären dann nicht mehr möglich. So gesehen ist die Verwendung von XForms-Einträgen in GUIDD-Dateien zumindest für GUI-Elemente keine schlechte Wahl. Für Elemente mit komplexem Typ (Container) sowie die Informationen zu Dienst und Operation bietet XForms jedoch keine passenden Syntaxelemente an, so dass an der Stelle noch Änderungen zu erwarten sind.

7.2.5. XML DOM

Auch auf unterster Ebene im Bereich der XML-Standards sind Verbesserungen vorstellbar. Eine im Laufe der praktischen Arbeit aufgetretene Art von Serialisierungsfehlern von XML-Dokumenten mit Namensräumen führte zu dieser Erkenntnis. Das Problem liegt darin begründet, dass der Zugriff auf XML in Baumstruktur, die sogenannte *DOM*-Schnittstelle (*Document Object Model*), vorsieht, bei Verwendung von Namensräumen für Elemente und Attribute den voll qualifizierten Namen zu übergeben, also den lokalen Element- oder Attributnamen und einen Präfix. Dies kann jedoch zu Inkonsistenzen und in der Folge zu fehlerhaftem XML führen. Der an die Java-Syntax angelehnte Pseudocode von DOM kann das Problem verdeutlichen:

```
// Erstellt Attribut att1 mit Namensraum urn:ns1
element.createAttributeNS("urn:ns1", "x:att1", "...")
...
// Erstellt Attribut att2 mit Namensraum urn:ns2
element.createAttributeNS("urn:ns2", "x:att2", "...")
```

Ein Element mit zwei Attributen, die beispielsweise aufgrund einer Verarbeitung in unterschiedlichen Applikationen den gleichen Präfix für unterschiedliche Namensräume verwenden, kann prinzipiell erst einmal erzeugt werden. Die Serialisierung des DOM in ein XML-Dokument führt dann aber zu einem ungültigen Dokument, da die Attribute zur Deklaration des Namensraumes mehrfach vorhanden sind:

```
<element xmlns:x="urn:ns1" xmlns:x="urn:ns2" .../>
```

Die DOM-Spezifikation geht auch an einer Stelle auf das Problem ein:

> „In particular, since the mapping between prefixes and namespace URIs is not enforced, in general, the resulting document cannot be serialized naively."

Das Problem wäre jedoch umgehbar, wenn nur lokale Namen ohne Präfix, aber mit Angabe des Namensraumes, übergeben werden würden. Die XML-Serialisierungsroutine könnte optional Hinweise auf übliche Präfixvarianten wie `xsd` für XML Schema oder `xf` für XForms erhalten, es spricht jedoch nichts gegen völlig automatisch generierte Präfixangaben. Eine automatische Generierung mit einer erweiterbaren Sammlung an Präfixangaben würde darüberhinaus den Wildwuchs eindämmen, der beispielsweise bei XML Schema mit den Varianten `xsd`, `xs` und `s` präsent ist.

Eine solche XML-Verarbeitung ohne die Gefahr von den gezeigten Inkonsistenzen ist in der vom Autor erstellten Bibliothek XSD4J [73] implementiert.

7.2.6. Übersetzungskataloge

Keines der drei verbreiteten Formate Gettext (`po`-Kataloge und `pot`-Vorlagen), Java-Properties (`properties`-Dateien) und Qt-Übersetzungen (`ts`-Dateien) beinhaltet Metadaten zur Quell- oder Zielsprache. Gettext bietet zwar einen Kopfteil mit Metadaten an, dieser enthält jedoch nur Angaben zur Dateikodierung, zum Übersetzer und zum Projekt.

Eine Ergänzung dieser Datenfelder ist für eine automatisierte Einbindung der Übersetzungen unerlässlich, da sich die verarbeitenden Anwendungen nicht auf Konventionen zu Kommentaren, Dateinamen oder ähnlichem verlassen können. Für Gettext wird die Einführung der Felder `Source-Language` und `Target-Language` vorgeschlagen. Qt-Kataloge sollten ob ihrer Ablage als XML-Dateien Gebrauch des Attributs `xml:lang` machen und könnten somit sogar mehrere Übersetzungen pro Eintrag aufnehmen, so wie dies auch bei GUIDD-Dateien der Fall ist. Java-Property-Dateien benötigten spezielle ID-Namen, als Beispiel werden die Namen `Locale.Source` und `Locale.Target` vorgeschlagen.

Exemplarisch wird nun eine Qt-Übersetzungsdatei mit den vorgeschlagenen Änderungen gezeigt:

```
<!DOCTYPE TS><TS version="1.1+improved">
<context>
    <name>WSGUI</name>
    <message>
        <location filename="flight.wsdl" line="200"/>
        <source xml:lang="en">Reservation</source>
        <translation xml:lang="de">Reservierung</translation>
    </message>
</context>
</TS>
```

A. Beispieldateien

WSDL-Beispieldatei: Flugbuchungsdienst

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions
        xmlns:s="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:types>
    <s:schema elementFormDefault="qualified">

      <s:element name="ReservationResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="success"
               type="s:boolean" />
            <s:element minOccurs="0" maxOccurs="1"
               name="flightinformation" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="ReservationRequest">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="5"
               name="reservation" type="Reservation" />
            <s:element name="id" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>

      <s:complexType name="Reservation">
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="name"
             type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="seatclass"
             type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:schema>
  </wsdl:types>
```

```
  <wsdl:message name="ReservationRequestMessage">
    <wsdl:part name="parameters" element="ReservationRequest" />
  </wsdl:message>
  <wsdl:message name="ReservationResponseMessage">
    <wsdl:part name="parameters" element="ReservationResponse" />
  </wsdl:message>

  <wsdl:portType name="ReservationPort">
    <wsdl:operation name="Reservation">
      <wsdl:documentation>
        Reserve a flight.
      </wsdl:documentation>
      <wsdl:input message="ReservationRequestMessage" />
      <wsdl:output message="ReservationResponseMessage" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="ReservationSoap" type="ReservationPort">
    <soap:binding
       transport="http://schemas.xmlsoap.org/soap/http"
       style="document" />
    <wsdl:operation name="Reservation">
      <soap:operation soapAction="#Reservation" style="document"
         />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="FlightReserver">
    <wsdl:documentation>
      Flight Reservation System.
    </wsdl:documentation>
    <wsdl:port name="ReservationPort" binding="ReservationSoap">
      <soap:address location="http://localhost:10080/" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

GUIDD-Beispieldatei zur Flugbuchung

```
<!DOCTYPE guidd>
<deployment
        xmlns="http://wsgui.berlios.de/namespaces/guidd"
        xmlns:xf="http://www.w3.org/2002/xforms">
```

```
<operations>
  <operation name="Reservation">
    <prettyName>Reservierung</prettyName>
    <description>Fuehrt eine Reservierung durch</description>
    <submit>
      <xf:submit>
        <xf:label>Reservieren...</xf:label>
      </xf:submit>
    </submit>
  </operation>
</operations>

<formComponents>

  <!-- ixpath used here (instance-based matches) -->

  <formComponent ixpath="/ReservationRequest">
    <xf:output>
      <xf:label>Reservierungswunsch</xf:label>
    </xf:output>
  </formComponent>
  <formComponent ixpath="/ReservationRequest/reservation">
    <xf:output>
      <xf:label>Reservierung</xf:label>
    </xf:output>
  </formComponent>
  <formComponent ixpath="/ReservationRequest/id">
    <xf:secret>
      <xf:label>Identifikation</xf:label>
      <xf:hint>Passwort hier eingeben</xf:hint>
    </xf:secret>
  </formComponent>

  <!-- xpath used here (schema-based matches) -->

  <formComponent xpath="/Reservation/name">
    <xf:input>
      <xf:label>Name</xf:label>
      <xf:hint>Vor- und Zuname des Passagiers</xf:hint>
    </xf:input>
  </formComponent>
  <formComponent xpath="/Reservation/seatclass">
    <xf:input>
      <xf:label>Sitzklasse</xf:label>
      <xf:hint>Sitzklassen: economy, business oder
         firstclass</xf:hint>
    </xf:input>
  </formComponent>
```

```
  </formComponents>
</deployment>
```

SOAP-Datenübertragung zur Flugbuchung

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/1999/XMLSchema" >
  <SOAP-ENV:Body>
    <ReservationRequestMessage>
      <parameters>
        <reservation>
          <name xsi:type="xsd:string" >Josef Spillner</name>
          <seatclass xsi:type="xsd:string" >economy</seatclass>
        </reservation>
        <id xsi:type="xsd:string" >geheim</id>
      </parameters>
    </ReservationRequestMessage>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

B. Screenshots

Die Bilder B.1 und B.2 enthalten den Dienst- und den Nachrichteneditor. Im Diensteditor wurde anstelle der Baumansicht die Strukturansicht gewählt.

Screenshot B.3 zeigt das Programm *Kung* bei einem dynamischen Aufruf eines Webservices für Flugbuchungen. Unschwer zu erkennen sind die völlig unadäquaten Beschriftungen der Eingabefelder.

In Bild B.4 hingegen wurde eine GUIDD-Datei erstellt und für den Aufruf verwendet. Es können dadurch sprachabhängige Bezeichner und Hinweise als Tooltips verwendet werden. Desweiteren ist die besondere Behandlung von bestimmten Eingabefeldern beispielsweise für Passwörter möglich.

Die Ansicht im Dynvoker als alternative Aufrufvariante ist schließlich in Bild B.5 zu sehen. Die GUI ist sichtbar der des Kung-Aufrufes ähnlich. Man erkennt auch die Möglichkeit, durch Inferenzprinzipien Listenelemente hinzufügen zu können.

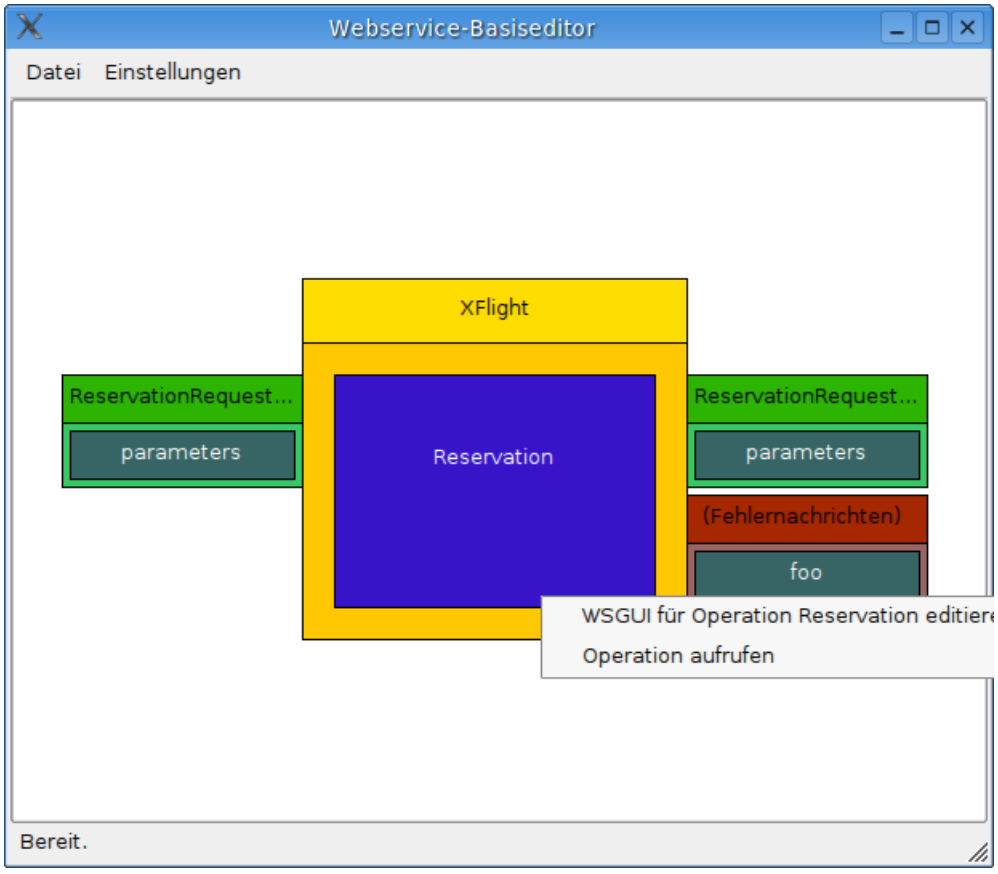

Abbildung B.1.: Strukturansicht im Diensteditor

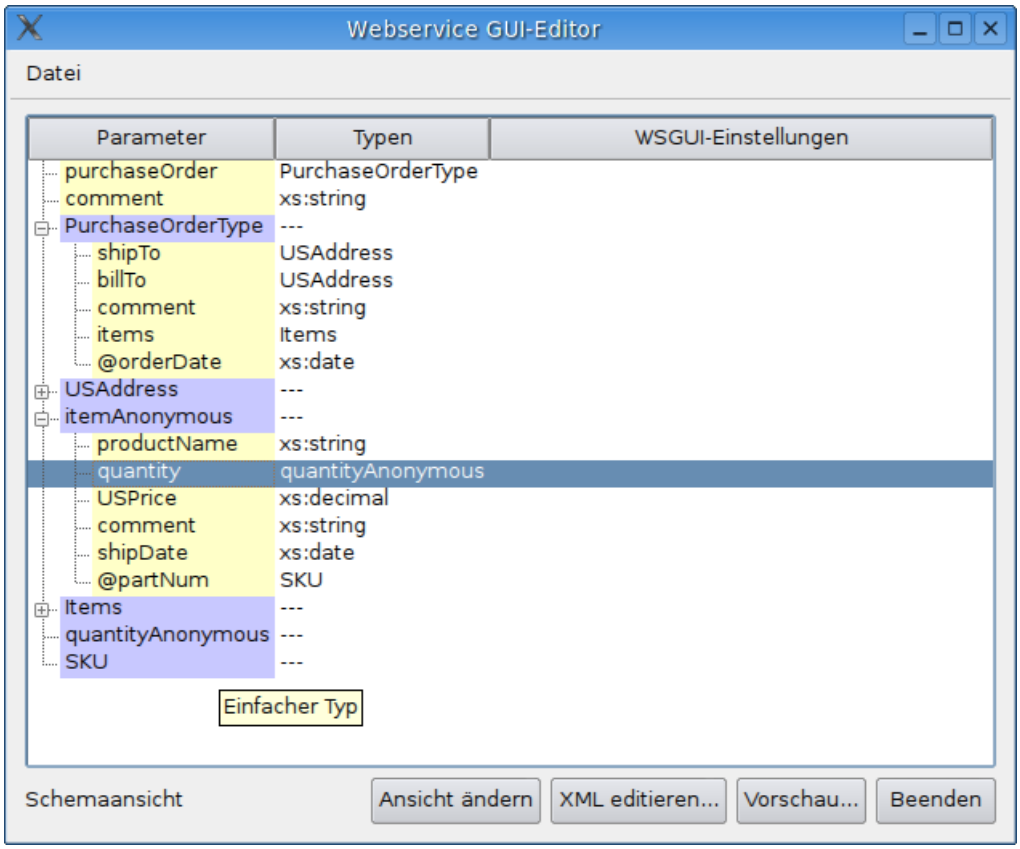

Abbildung B.2.: Baumansicht im Nachrichteneditor

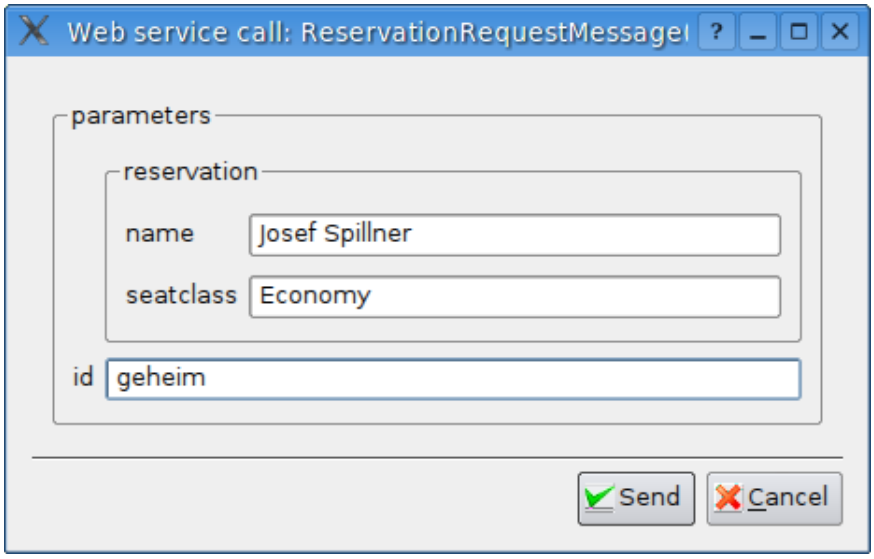

Abbildung B.3.: Webservice-Aufruf mit Kung (reine WSDL-Nutzung)

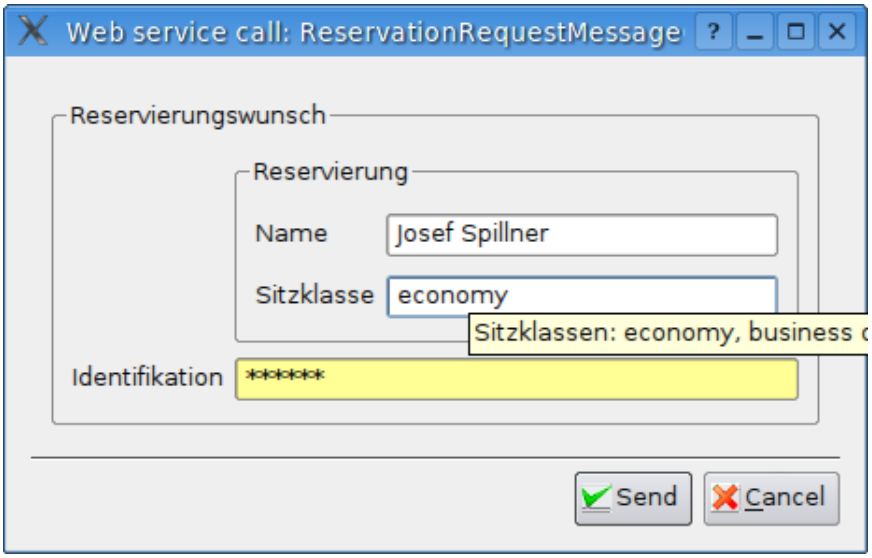

Abbildung B.4.: Webservice-Aufruf mit Kung (mit WSGUI-Informationen)

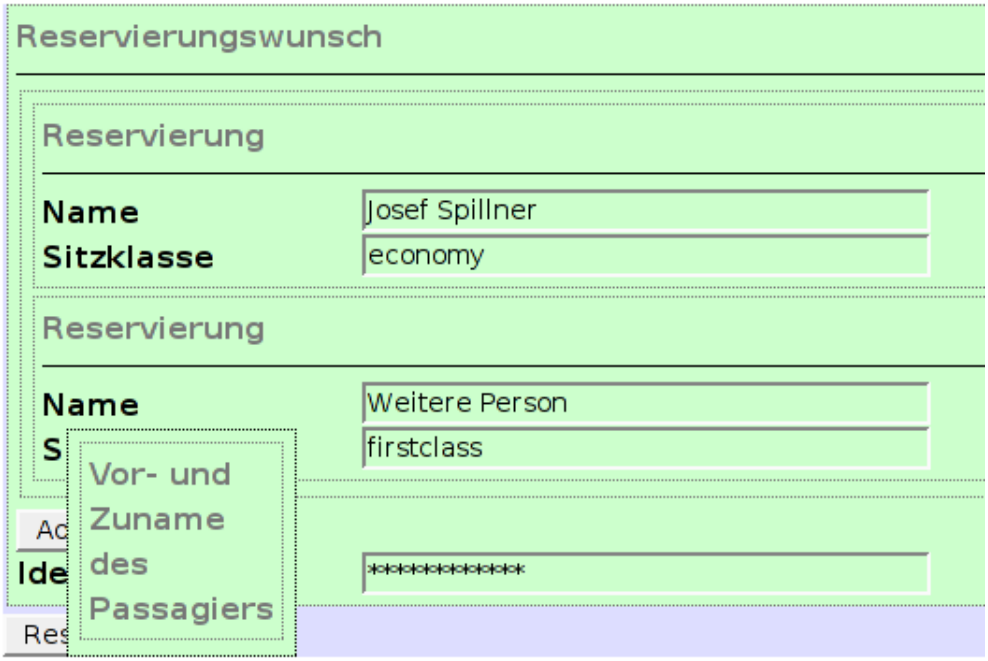

Abbildung B.5.: Webservice-Aufruf mit Dynvoker (mit WSGUI-Informationen)

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Aktas, Mehmet S. *et al.*: *Web Service Information Systems and Applications*, 2005. Indiana University.

[2] Anuff, Ed: *New Standard Closes the Loop for Web Services.* EAI Journal, Seiten 16–17, November 2001.

[3] authors, Various: *Widget*, June 2006. `http://de.wikipedia.org/wiki/Widget_(GUI)`.

[4] Baskerville, Richard *et al.*: *Extensible Architectures: The strategic value of service-oriented architecture in banking*, 2004. Case study.

[5] Beeger, Robert F.: *Herstellung von Konsistenz und Validität in Web-Anwendungen.* Diplomarbeit, Universität Hamburg, 2003.

[6] Beinhauer, Wolfgang und Thomas Schlegel: *User Interfaces for Service Oriented Architectures.* In: *Proceedings of the 11th International Conference on Human-Computer Interaction*, July 2005. HCI International 2005.

[7] Bex, Geert Jan, Frank Neven und Jan Van den Bussche: *DTDs versus XML Schema: A Practical Study.* In: *Proceedings*, June 2004. Seventh International Workshop on Web and Databases (WebDB 2004), Paris, France.

[8] Biel, Bettina und Volker Gruhn: *Towards Usability-Improving Design Patterns for Mobile Client-Server Computing*, June 2006. Universität Leipzig, work-in-progress paper for EuroPLoP 2006.

[9] Bleyh, Nicolás: *Analyse und Vergleich von Ansätzen zur Einbindung von menschlichen Interaktionen in komplexe Web Services.* Diplomarbeit, TU Dresden, June 2006.

[10] Boddington, Richard, Judy Clayden, Michael Collins und Sam Pride: *A multi-lingual web-based survey form machine translation mechanism.* In: *Proceedings*, 2003. First Australian Undergraduate Students' Computing Conference, Perth, Australia.

[11] Boyer, John M., David Landwehr, Roland Merrick, T.V. Raman *et al.*: *XForms 1.0 (Second Edition)*, 1.0 Auflage, March 2006.

[12] Bradner, Scott O.: *The Internet Standards Process*, Revision 3 Auflage, October 1996. RFC 2026/BCP 9.

[13] Bray, Tim *et al.*: *Extensible Markup Language (XML)*, 1.1 Second Auflage, August 2006.

[14] Bresnahan, Timothy F. und Pai-Ling Yin: *Economic and Technical Drivers of Technology Choice: Browsers*, 2005. Draft.

[15] Chinnici, Roberto, Jean-Jacques Moreau, Arthur Ryman und Sanjiva Weerawarana: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, Candidate Recommendation Auflage, March 2006.

[16] Christensen, Erik, Francisco Curbera, Greg Meredith und Sanjiva Weerawarana: *Web Service Description Language (WSDL)*, 1.1 Auflage, March 2001.

[17] Clark, James: *XSL Transformations (XSLT)*, 1.0 Auflage, November 1999.

[18] Clark, James: *TREX - Tree Regular Expressions for XML*, 1.0 Auflage, February 2001.

[19] Clark, James und Steve deRose: *XML Path Language (XPath)*, 1.0 Auflage, November 1999.

[20] Clinton, DeWitt *et al.*: *OpenSearch 1.1 Description*, Draft 3 Auflage, September 2006.

[21] developers, Axis: *Apache Axis*, September 2006. `http://ws.apache.org/axis/`.

[22] developers, Eclipse: *Eclipse IDE and Editor Framework*, September 2006. `http://www.eclipse.org/`.

[23] developers, KDE: *K Desktop Environment*, September 2006. `http://www.kde.org/`.

[24] Eberleh, Edmund: *Beschreibung, Klassifikation und mentale Repräsentation komplexer Mensch-Computer-Interaktion*. Theorie und Forschung - Psychologie. S. Roderer-Verlag Regensburg, 1989.

[25] Falk, Steffi: *Algorithmus zur Ähnlichkeitsbestimmung von XML-Dokumenten und -Schemata*. Diplomarbeit, Universität Rostock, 2005.

[26] Fallside, David C.: *XML Schema Part 2: Datatypes (Second Edition)*, 1.0 Auflage, October 2004.

[27] Feigenbaum, Barry und Michael Squillace: *Reflexive User Interface Builder*, February 2006.

[28] Gehrmann, Bernd: *KDE Architecture Overview*, 2002. Includes KIO.

[29] George, Thomas und Gerd Westerman: *Model-driven Architecture in einer serviceorientierten Anwendungslandschaft*. JavaSpektrum, 1(1):18–21, January 2006.

[30] Godwin-Jones, Robert: *Ajax and Firefox: New Web Applications and Browsers.* Language Learning and Technology, 9(2):8–12, May 2005.

[31] Hallgren, Thomas und Magnus Carlsson: *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces.* Dissertation, Göteborg University, Department of Computing Science, 1998.

[32] Hately, Andrew, Luc Clement, Claus von Riegen und Tony Rogers (eds): *UDDI Spec Technical Committee Draft*, 3.0.2 Auflage, October 2004.

[33] Helmolt, Arnim von: *Automatische Generierung von Update-Masken aus Datenbanksichten.* Diplomarbeit, Universität Hannover, 1999.

[34] Hendry, Kai: *Web engineering for mobile devices.* Diplomarbeit, University of Helsinki, Dept. of Computer Science, February 2005.

[35] Hickson, Ian: *Web Forms 2.0*, 2.0 Working Draft Auflage, August 2006.

[36] Honkala, Mikko und Petri Vuorimaa: *A Configurable XForms Implementation.* In: *Proceedings*, December 2004. IEEE Sixth International Symposium on Multimedia Software Engineering (ISMSE'04).

[37] IBM: *Web Service eXperience Language (WSXL)*, 2.0 Auflage, April 2002.

[38] Isys: *Hall of Shame of User Interfaces*, June 2006. `http://homepage.mac.com/bradster/iarchitect/shame.htm`.

[39] James Clark, MURATA Makoto: *RELAX NG Specification*, Committee Specification Auflage, December 2001.

[40] Jouanneau, Laurent: *Mozilla et XUL: Interfaces riches dans les applications web*, April 2004. Libr'east 2004, Paris, France.

[41] Kassoff, Michael, Daishi Kato und Waqar Mohsin: *Creating GUIs for Web Services.* IEEE Internet Computing, 7(4):66–73, September/October 2003.

[42] Kassoff, Michael, Josef Spillner *et al.*: *GUIDD: GUI Deployment Descriptor file format specification*, 0.99.5 Auflage, May 2006.

[43] Kelly, Kevin E., Jan Joseph Kratky, Keith Wells und Steve Speicher: *XML Forms Generator*, March 2006. IBM alphaWorks.

[44] Kirtland, Mary: *A Platform for Web Services*, January 2001. Microsoft Developer Network.

[45] Korotkiy, Maksym: *Towards an Ontology-enabled Service-Oriented Architecture*, 2005. Vrije Universiteit Amsterdam.

[46] Kraut, Sophie: *Konzeption und Implementierung eines Tools in Java zum Entwurf von Masken und Generieren von XML als Ablageformat.* Diplomarbeit, Fachhochschule Mannheim, 2003.

[47] Krishnan, Sriram, Brent Stearn, Karan Bhatia, Kim K. Baldridge, Wilfred Li und Peter Arzberger: *Opal: Simple Web Services Wrappers for Scientific Applications.* Technischer Bericht, San Diego Supercomputer Center, March 2006.

[48] Kruschinski, Volker: *Layoutgestaltung grafischer Benutzungsoberflächen.* Dissertation, Ruhruniversität Bochum, 1999.

[49] Lay, Patrick und Stefan Lüttringhaus-Kappel: *Transforming XML Schemas into Swing GUIs*, 2004. Universität Bonn.

[50] Liam O'Brien, Len Bass, Paulo Merson: *Quality Attributes and Service-Oriented Architectures.* Technischer Bericht, Carnegie-Mellon University, September 2005.

[51] Luyten, Kris: *UIML - The User Interface Markup Language*, December 2004. IMEC Workshop on standards in Multimedia, Leuven, Belgium.

[52] Marinelli, Paolo, Claudio Sacerdoti Coen und Fabio Vitali: *SchemaPath, a Minimal Extension to XML Schema for Conditional Constraints.* In: *Proceedings*, May 2004. WWW2004.

[53] McLaren, Chris *et al.*: *Sweet Project – XSWT Requirements and Design*, August 2003. Proposal.

[54] Mercedes, Pilar Hernández: *En torno a la traducción automática.* Cervantes, 1(2), March 2002.

[55] Mitra, Nilo: *SOAP Version 1.2 Part 0: Primer*, 1.2 Auflage, June 2003.

[56] Mohammed, Sabah, Jinan Fiaidhi, Hamada Ghenniwa und Marshall Hahn: *Developing a Secure Web Service Architecture for SVG Image Delivery.* Journal of Computer Science, 2(2):171–179, February 2006.

[57] Myers, Brad A.: *A Brief History of Human Computer Interaction Technology*, 1996. Carnegie Mellon University.

[58] Mönnich, Uwe: *Sonderforschungsbereich 441: Linguistische Datenstrukturen*, January 2001. Antrag für die zweite Phase 2002 - 2004.

[59] Nilson, Jörn: *N-GUIDE – ein System zur Erstellung grafischer Benutzeroberflächen.* Dissertation, Technische Fakultät der Universität Erlangen-Nürnberg, 1998.

[60] Pemberton, Steven und others (W3C): *The Extensible HyperText Markup Language*, 1.0 Revised Auflage, August 2002.

[61] Pemberton, Steven und others (W3C): *XHTML 2.0*, Working draft Auflage, July 2006.

[62] Pierce, Lyn: *Traits UI User Guide*, 2005. Draft.

[63] Rajasekaran, Preeda: *Enhancing Web Service Descriptions Using WSDL-S*. Diplomarbeit, University of Georgia, Computer Science Department, 2005.

[64] Reis, Christian Robottom und Renata Pontin de Mattos Fortes: *An Overview of the Software Engineering Process and Tools in the Mozilla Project*, February 2002.

[65] Rosenberg, Linda H. und Larry Hyatt: *Applying and Interpreting Object Oriented Metrics*, April 1998. NASA SATC talk at Software Technology Conference, Utah, USA.

[66] Schade, Norbert: *Universal Printer Description Format (UPDF)*, 1.0 (working draft) Auflage, 2004.

[67] Schlee, Max und Jean Vanderdonckt: *Generative Programming of Graphical User Interfaces*, 2004. PoLITe project.

[68] Schumacher, Cornelius: *KXForms - automatic configuration dialog generation*, August 2005. aKademy conference.

[69] Schumacher, Cornelius, Tobias König *et al.*: *KDE PIM Kode Framework*, September 2006. `http://pim.kde.org/`.

[70] Schönböck, Johannes: *Erstellung eines MDA-Editors zur Generierung von Applikationsrahmen für mobile Endgeräte.* Diplomarbeit, FH Hagenberg, June 2005.

[71] Simon, Rainer, Florian Wegscheider und Konrad Tolar: *Tool-Supported Single Authoring for Device Independence and Multimodality.* In: *Proceedings*, September 2005. MobileHCI'05, Salzburg, Austria.

[72] Spillner, Josef: *Inference: From XSD to XForms.* Unpublished, August 2006.

[73] Spillner, Josef: *Project Dynvocation*, May 2006. Student project at `http://dynvocation.selfip.net/`.

[74] Spillner, Josef: *WSGUI concept project*, January 2006. `http://wsgui.berlios.de/`.

[75] Spillner, Josef, Nicolás Bleyh *et al.*: *Dynvoker web service invocation servlet*, August 2006. Student project.

[76] Spillner, Josef, Iris Braun und Alexander Schill: *Flexible Human Service Interfaces*, July 2006. Submitted for publication.

[77] Spillner, Josef, Iris Braun und Alexander Schill: *WSInterConnect: Dynamic Composition of Web Services through Web Services.* In: *Lecture Notes in Computer Science (LNCS) 4025*, June 2006. Distributing Computing Techniques (DisCoTec)/DAIS'06, Bologna, Italy.

[78] System, SQLData: *SOAPclient*, 2005. `http://soapclient.com/soaptest.html`.

[79] TC, OASIS UIML: *User Interface Markup Language (UIML) Specification*, 3.1 Auflage, 2004.

[80] XMethods: *XMethods*, 09 2006. Web Services broker.